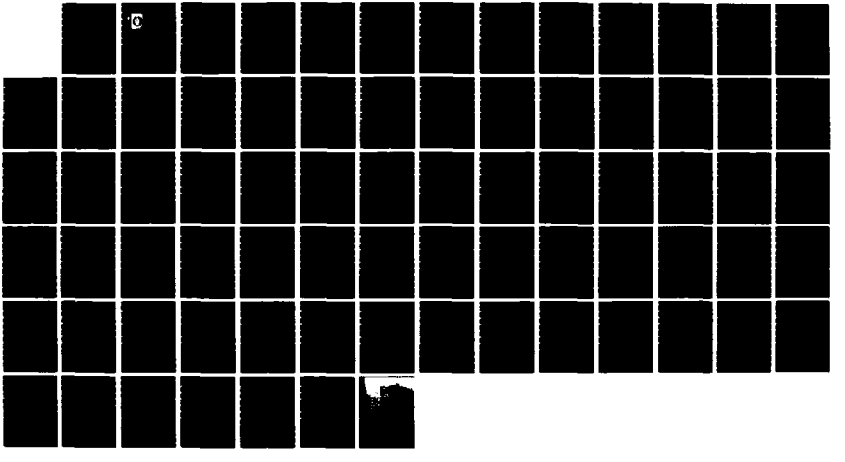
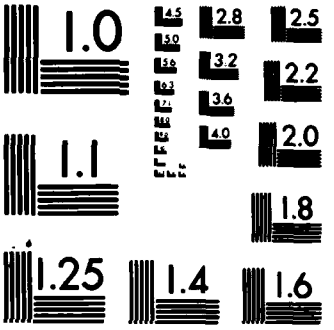
AD-A140 363 INTERMEDIATE ACCESS SWITCH: A MULTIPROCESSOR APPLICATION(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) T G CONNELLY ET AL. SEP 83 1/1

UNCLASSIFIED RSRE-83004 DRIC-BR-91262 F/G 9/1 NL





(2)

UNLIMITED

Report No. 83004

ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

INTERMEDIATE ACCESS SWITCH – A MULTIPROCESSOR APPLICATION

Authors: T G Connelly and M P Griffiths

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

REPORT 83004

Title: INTERMEDIATE ACCESS SWITCH - A MULTIPROCESSOR APPLICATION

Authors: T G Connelly and M P Griffiths

Date: September 1983

SUMMARY

Design options for the implementation of the control software for a computer controlled, circuit switching exchange (Intermediate Access Switch or IAS) on a multi-processor system (DISCUS) are outlined. The impact of DISCUS and its operating system on software structure and software development, integration, and testing methods is discussed, and a set of guidelines for the division of a complete software package among several co-operating processors is drawn up. The advantages and penalties of the multi-processor approach, as compared with the single processor approach, are pointed out, and a quantitative assessment of system performance based on the use of a traffic generator is presented.

CONTENTS

APPENDICES

1 INTRODUCTION

The work described here concerns the development of the so-called Intermediate Access Switch (IAS) at RSRE Malvern. IAS is a computer controlled, circuit-switched communications exchange providing most of the facilities required of a switch in a tactical area network.

IAS was developed primarily as a fairly large and complex application for the DISCUS multi-processor hardware scheme[2,7] and the DISCUS operating system[1]. We are therefore mainly concerned, in this report, with the control element of the switch. The switch hardware is described elsewhere[3,4,5].

The aims of the project were fourfold:

(1) To determine whether the DISCUS hardware and operating system were viable for use in a fairly large system.

(2) To establish the advantages and disadvantages of implementing switch control software in the environment provided by DISCUS. As far as software design is concerned, this environment is provided by the DISCUS operating system, which makes the DISCUS multi-processor appear, to the programmer, much like a multi-processing system running on a single processor. Thus many of the results obtained under this heading apply equally to single processor and multi-processor systems. However, the multi-processor architecture offers a number of facilities not available from single processor systems (such as parallel processing, and the ability to increase throughput by adding more processors running the same program) and results relating to the use of these facilities apply only to DISCUS and any similar multi-processor systems.

(3) To draw up a list of guidelines indicating how a large body of software for this type of application, or indeed any other, should be split up and distributed among the processors of a multi-processor system. Again, many of these guidelines apply equally well to multi-processing systems running on a single processor.

(4) Eventually to produce a number of switches which could be connected together to form a small communications network, thus providing a tool for research into the characteristics and behaviour of this type of network.

2 OUTLINE OF DISCUS SYSTEM

In order to fully understand the discussion of IAS software given in section 4, a certain amount of knowledge of the DISCUS system is required. This section therefore presents a brief outline of both the hardware and software structure of the DISCUS system. Those familiar with this material may thus proceed direct to section 3.

2.1 Hardware

The structure of a single DISCUS computer is illustrated in Figure 2.1. This shows the processor working via its own local bus to its program code and its local work space in RAM. Access to the outside world is achieved by the use of a peripheral interface which may go to, say, a VDU or some other more specialised applications hardware.

An extra feature is the bus supervisor which allows observation of activity on the bus via a display device (e.g. a logic analyser) and also performs a check on every transaction on the bus. If a bus transaction fails, that is to say the addressed memory or device did not respond within a specified time interval, the bus supervisor will light a fault lamp and also, depending on which option has been selected, either:

(a) Halt the processor.

(b) Apply a default response to allow the processor to continue,

or

(c) Apply a default response and interrupt the processor to a fault routine

Option (c) is selected for the current version of the operating system.

The bus supervisor also has four switches on it which provide a unique four bit processor identity which is used by the operating system to distinguish between the processors. Incidentally, this is, at the moment, why we are limited to 15 processors per system as the setting all zeroes is reserved for operating system use.

Remember that all this activity takes place totally separately from any other activity on any other bus, and therefore any computer may take recovery action without disturbing the rest of the system. Also note that each computer may only access its own local store, peripherals and global store and is physically incapable of directly accessing any other computer's local store or peripherals. When a computer decides that it needs to access global store, it asynchronously requests the use of the global bus via the crate bus interface.

For reasons of convenience and in order to reduce the number of interconnecting wires, computers are packed two or three to a 19" rack or crate. The structure of a crate is illustrated in Figure 2.2 and shows (in this case) three computers accessing their local crate bus. Arbitration of requests between the computers is done by the crate bus controller which then in turn sends a request to the global store itself.

The global store fits into a crate by itself and is illustrated in Figure 2.3. This shows each of the crates accessing the global store bus via its own global store interface. Arbitration of requests by each crate is taken care of by the global store controller which, when it has decided which crate has

access to the store, sends a "grant" signal back down the appropriate interface to allow the local processor to proceed. Global store is partitioned into up to 7 blocks of 32k bytes (PROM or RAM) and the desired block is selected by each processor when it makes its request in a manner which is described in the following paragraph.

The address bus is extended from its normal 16 bits to 19 bits by the use of a register residing on each processor. This is called the Auxiliary Address Register (AAR) and writing a non-zero value to this constitutes a request for the appropriate block of global store. Each processor's address space then looks like Figure 2.4 with local 0 (its bottom 32k) always present and the upper 32k being local (with the AAR set to zero) or the appropriate block of global store in the range 1 to 7 (with the AAR set to a number from 1 to 7). Thus the AAR page bits are only used if the most significant bit of the "normal" address (i.e. the 16th) is a one, otherwise they are ignored and the request goes to local store.

Although the global store remains overlaid to a particular computer as long as the AAR is set, there is no guarantee that no other processor has written to it between the first processor's accesses. Therefore, in order to implement indivisible (read - modify - write) operations, an additional bit is provided in the AAR to specify that the whole global store will be locked, that is no other processor is allowed access to the global store while that bit is set. Each processor may also specify that global store access is to be read only or read/write in order to improve error detection.

There is one further modification to the address map of each processor: this is the introduction of the bootstrap PROM which resides on each processor board. The PROM is engaged by pushing the BOOT button on the front panel and this overlays the bottom 1k of the processor's local store. Reads come from the PROM, and writes go to the overlaid RAM (if RAM is present at these addresses). The PROM is disengaged by executing the instruction in, or reading from, its last location and can only be re-engaged by pushing the BOOT button again. The current version of the operating system uses it to contain the loader and code for the initialisation of global store.

The loader is a two part program; one part runs on the Intel Microcomputer Development System (MDS), extracts the compiled functions from the disc and places them in the global store. The other part is a small assembly language program in the bootstrap PROM of each of the DISCUS computers. These look at the global store and copy the appropriate function into their local RAM for execution. This approach is particularly nice as it means that the minimum amount of extra hardware is required, just the board to convert the MDS into a processor which can access the DISCUS global store.

2.2 Software

The processors in a DISCUS system communicate via the common, passive global store by placing data in selected areas of the store where it may be read by the receiving processor. This exchange of information is formalised by the use of a number of standard procedures which, due to the fact that processors local stores are only accessible to themselves, and not each other, are resident on each processor. This set of procedures is referred to as the Operating System. The operating system procedures manipulate a user defined set of areas of the global store (called global objects) in a standard manner in order to achieve an ordered flow of data between the processors. There are basically three classes of global object supported by the current version of the operating system:

(1) **Synchronisers**, which are the most basic objects available, and are designed for protection of global resources or construction of special applications semaphores where the more complex DISCUS methods are not appropriate.

(2) **Arrays**, which are collections of multi-byte objects accessed by an indexing mechanism, and come in two types which differ in the security of access given to the information in them. Both types are single dimensioned, with lower bound zero, the upper bound being set when they are declared. Although the arrays have only one index each element is not constrained to a single byte, but may have a declared width. When the element is accessed the whole <width> bytes are indivisibly transferred either to or from the user's program.

(3) **Channels**, which are first-in, first-out queues which are designed for passing transient (destructively read) data between functions, as opposed to the non-transient (destructively written) facilities provided by the arrays. Channels are single buffer, single direction queues and are used to pass messages or stimuli from one processor to another. They are consequently the primary means of initiating processing on a processor which has no inputs from the outside world (i.e. has no peripherals from which it can read). Thus most processors sit on an input channel waiting to be given something to do, returning to the channel for the next job when they have finished.

In order to make the use of these objects as easy as possible, the Coral language has been extended to allow the declaration of the objects within an ordinary program. Then, as part of the specially designed package of translator programs, known as the System Generator, a pre-pass program goes through the source file and converts the special DISCUS constructs into standard Coral before being processed by the compiler.

More detailed descriptions and instructions on how to use all of these features will be found in the operating system report[1].

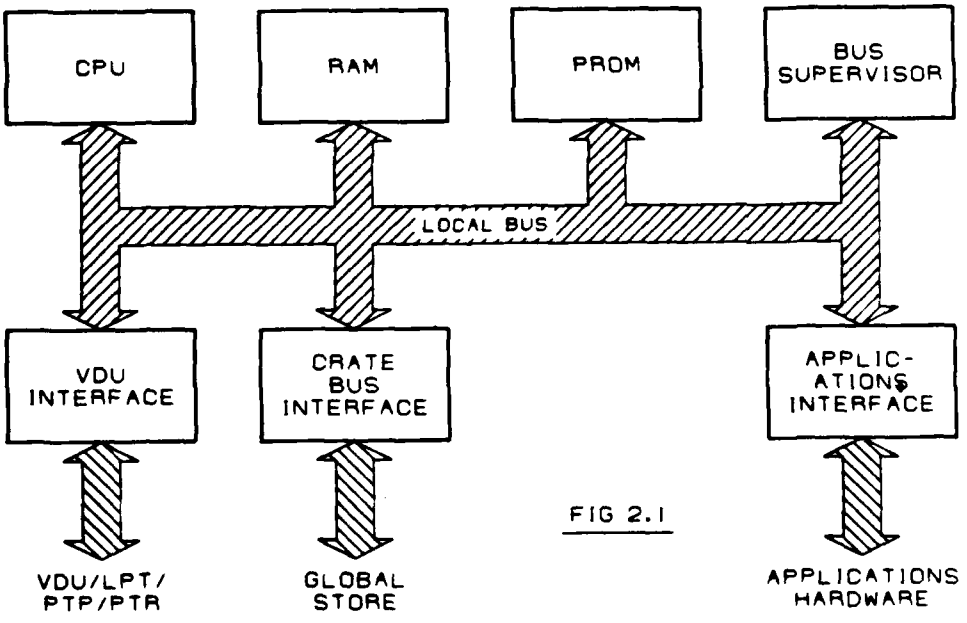


FIG 2.1

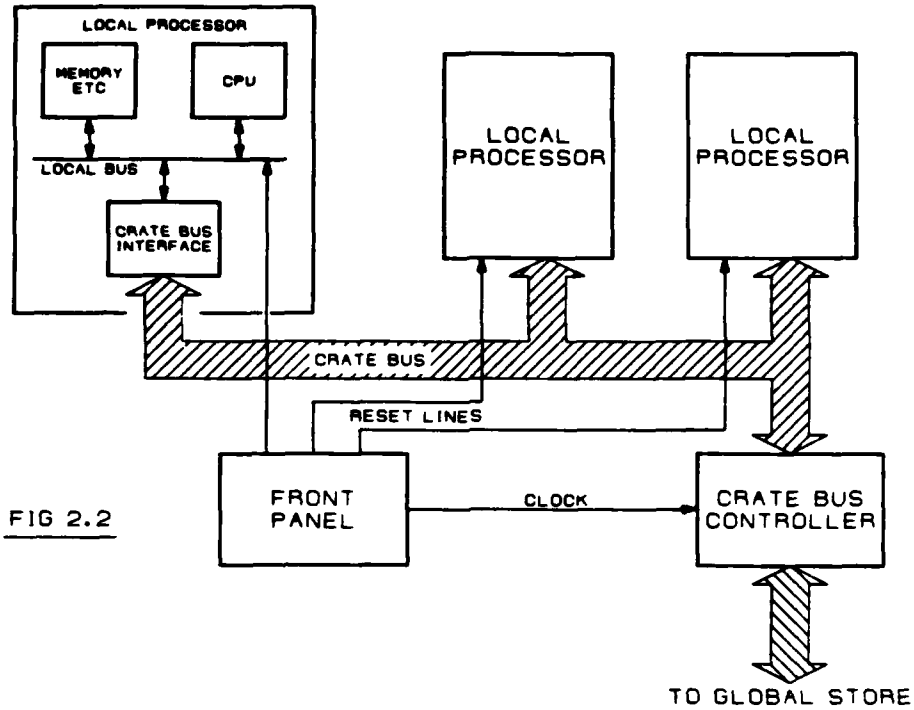

LOCAL PROCESSOR
MEMORY ETC
CPU
LOCAL BUS
CRATE BUS INTERFACE
LOCAL PROCESSOR
LOCAL PROCESSOR
CRATE BUS
RESET LINES
FRONT PANEL
CLOCK
CRATE BUS CONTROLLER
TO GLOBAL STORE

FIG 2.2

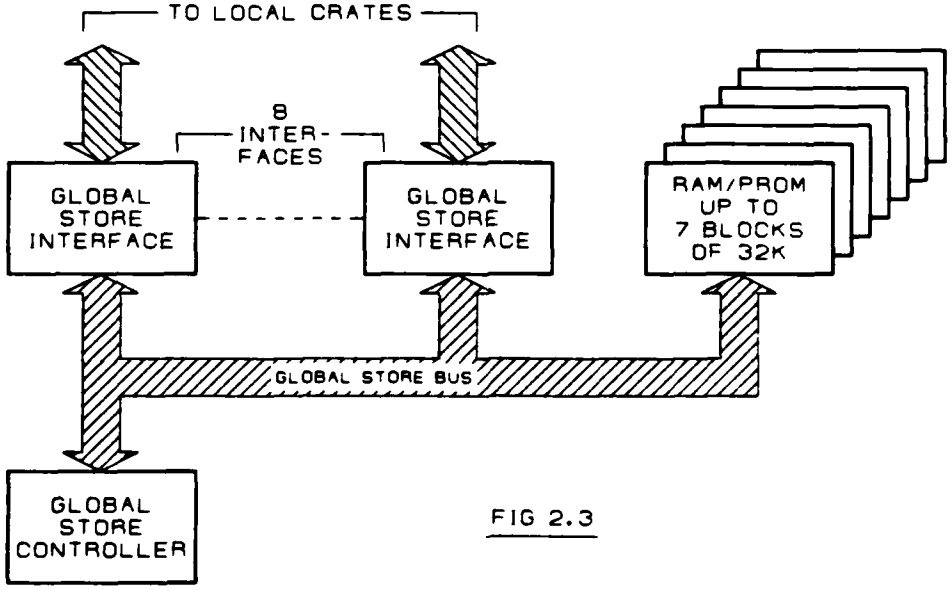

TO LOCAL CRATES
8 INTER-FACES
GLOBAL STORE INTERFACE
GLOBAL STORE INTERFACE
RAM/PROM UP TO 7 BLOCKS OF 32K
GLOBAL STORE BUS
GLOBAL STORE CONTROLLER

FIG 2.3

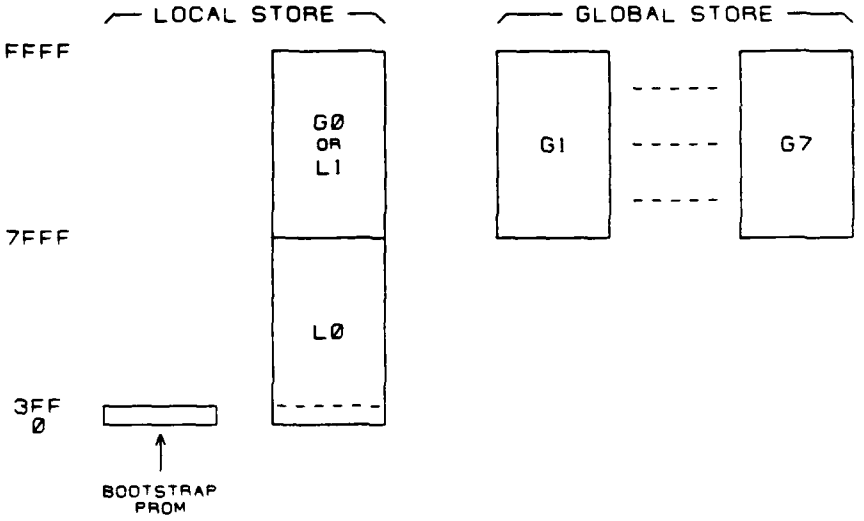


FIG 2.4 PROCESSOR ADDRESS SPACE

3 OUTLINE OF THE APPLICATION

Before proceeding further, a brief description of the IAS as a switching node in a communications network is necessary. This section therefore describes how the IAS is designed to fit into the network and the facilities that it provides to subscribers.

3.1 Communication Links

IAS is a computer controlled, circuit switching communications node. The node provides switching between up to five communication groups, two of which are normally designated to be trunk groups, and the remaining three local groups (see fig 3.1). The trunk groups provide communication links to remote switching nodes, and the local groups provide access to the network (of which IAS forms a part) for communities of subscribers, via a device called a Local Distribution and Access multiplexer (LDA).

A group, trunk or local, is a 512 kb/s data stream which carries information at 16 kb/s on each of 32 time division multiplexed (TDM) channels. Thus the data stream may be considered as a series of frames, each frame having 32 time slots, as shown in fig 3.2. In any group, channel 0 always carries a specific bit pattern (a 15 bit Pseudo Random Binary Sequence or PRBS). This identifies channel 0 and so enables the correct demultiplexing of the TDM bit stream. In a trunk group, channel 1 is designated to be the trunk signalling channel, of which more later. In a local group this channel carries no information. The remaining 30 channels of either type of group carry the traffic generated by the users of the network, plus, on local groups during call set-up and clear down, in band signalling.

3.2 Local Calls

A local group can support up to 30 subscribers, each of which is allocated a unique channel in the TDM frame. Local call set-up is achieved by a conversation between the subscriber terminal and the switch in which a series of 8 bit Cyclically Permutable Codes (CPC) are exchanged. These codes indicate to the switch that the subscriber has gone off-hook, dialled certain digits, and so on. When a subscriber has gone off-hook, and dialled a sequence of digits representing the identity of the subscriber with whom he wishes to communicate, the switch, having collected the digits, discovers from its data-base the group and channel on which the called subscriber is to be found. It then exchanges another series of CPCs with this terminal which results in the bell of the called subscribers terminal ringing. When this terminal goes off-hook in answer, the output channel of the calling terminal is connected to the input channel of the called terminal and vice-versa. The two terminals are then in traffic. When one of the terminals subsequently goes on-hook, the call is cleared down by breaking the connections between the channels.

From the above description it is apparent that some code detecting equipment is required, together with a code generator and a digital switching matrix. These are shown in fig 3.3. In addition, if the IAS is not to wait indefinitely for the arrival of an expected reply from a faulty or absent terminal, some means of timing a timeout for such waiting periods must be provided, that is, a real time clock of some sort is needed. All these peripheral devices must interface with the DISCUS hardware and the IAS applications software which runs on it. Details of these peripherals and their interfaces may be found elsewhere[3,4].

3.3 Trunk Calls

Trunk calls are originated in the same way as local calls. The difference lies in the fact that the dialled number identifies a subscriber who is not to be found on any of the local groups, but must be contacted via a trunk group. A trunk call is then set up using a trunk group which, according to the switch data-base, forms the first link in a route which will lead, via one or more other switches, to the wanted subscriber terminal. The mechanism by which the call is set up consists of the exchange of trunk messages (TMs) between the various switches on the chosen route. These messages are passed via channel 1 of the groups involved. When the message exchange is complete, the switching matrices in each of the switches concerned make connections between selected channels of their trunk groups until a complete bi-directional data link has been established between the calling and the called terminals. Clear down consists of a further trunk message being sent along the route which stimulates the breaking of the matrix connections and the freeing of the trunk channels. A typical trunk call set up is shown in fig 3.4.

The lower levels of the trunk signalling protocol, e.g. BCH error protection coding, information block numbering, automatic retransmission of corrupted blocks (RTM) etc, are controlled by a Trunk Signalling Unit (TSU), as shown in fig 3.3. A detailed description of the TSU and its interfaces to DISCUS can be found in the separate reports[3,5].

3.4 Data-base Management

The IAS requires a data-base which associates local subscriber identities with the communication channels by which they access the switch, and which describes the inter-switch connectivity. In a tactical area network, both subscribers and switches are mobile, and consequently the data-base must be continually updated to reflect changes in network configuration. Thus, in order to support its main job of setting up and clearing down calls, the IAS needs to obtain new information about the network of which it forms part, and incorporate this in its data-base.

3.5 User Facilities

In order to provide a sufficiently large and complex application to achieve the aims of the project as set out in section 1, the following user facilities (i.e. facilities available to subscribers at their terminals) were implemented in the IAS:

(1) Local call set up and clear down

(2) Trunk call set up and clear down

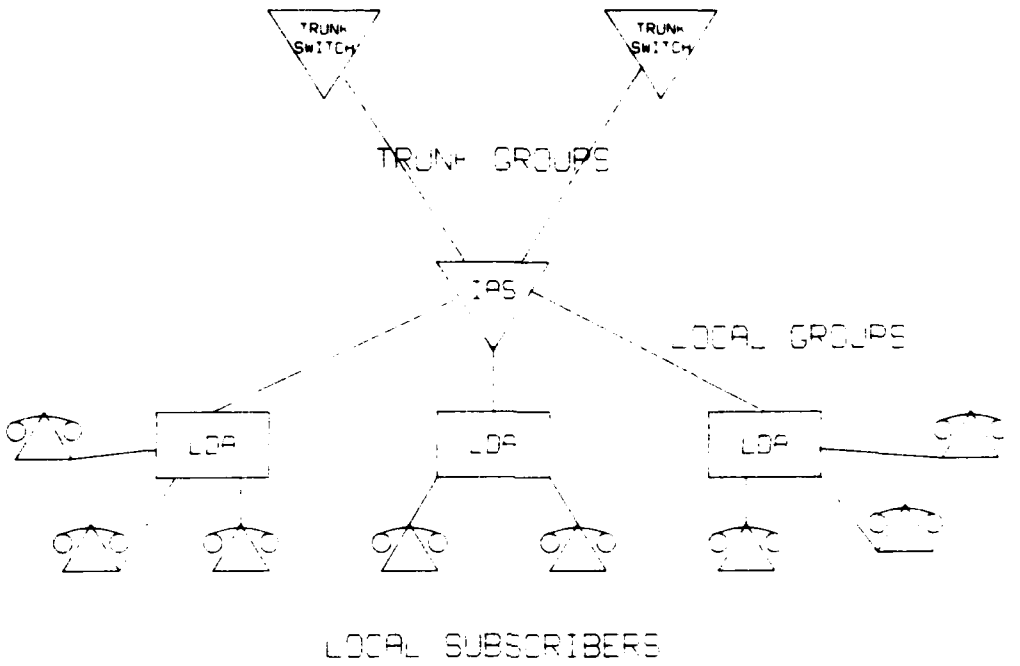
(3) Call hold - the facility for a subscriber to place an existing call to or from his terminal in a kind of pending state while he makes another call, and to subsequently return to the pending (or held) call without re-dialling the held subscriber's number.

(4) Call forward - the facility for a subscriber originating or terminating a two party call to cause a third subscriber's terminal to replace his terminal's part in the call, thereby removing his terminal from the call.

(5) Call transfer - the facility for a subscriber to instruct the system that all calls which would normally be terminated by his terminal are to be terminated by a different terminal until further notice.

(6) Pre-emption - the facility for a subscriber capable of making calls at a higher precedence level than that normally used by most subscribers to take over, where necessary, communications facilities currently in use by calls of a lower precedence.

FIG 3 1 IAS AND ITS NETWORK ENVIRONMENT



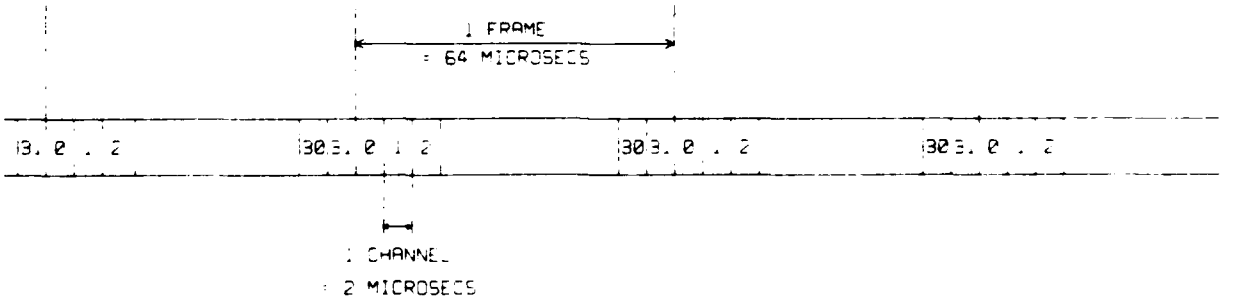


FIG 3 2 FRAME STRUCTURE OF A 512 KB/S T3 M LINK

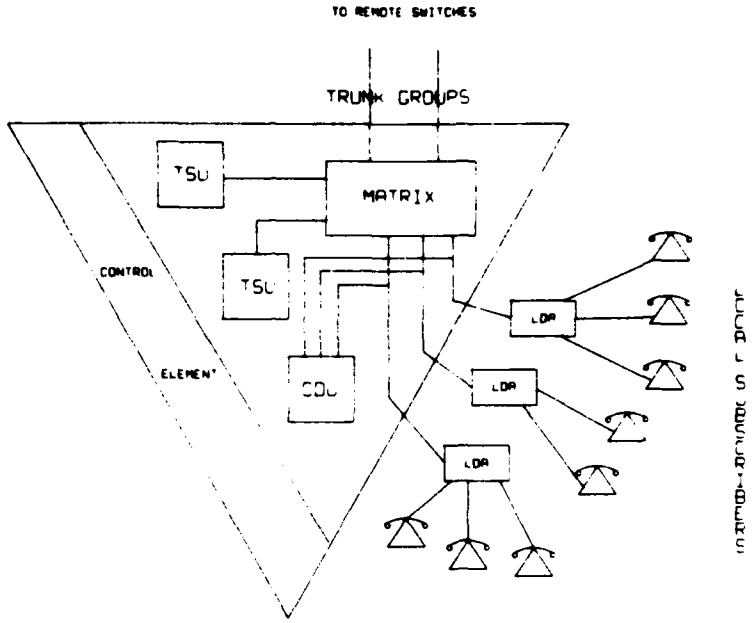


FIG 3 3 BASIC ELEMENTS OF IAS AND ITS LOCAL SUBSCRIBERS

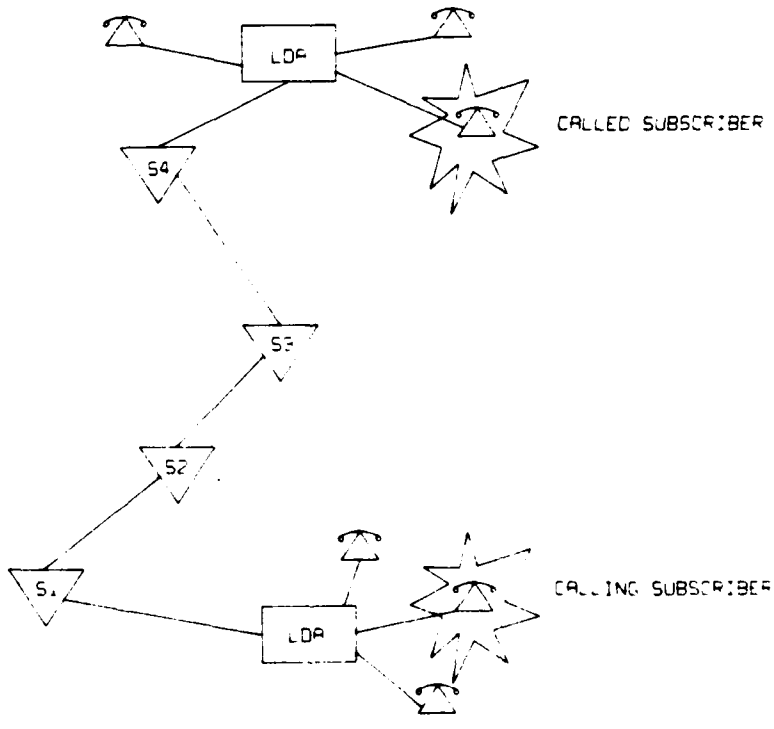


FIG 3 4 TYPICAL TRUNK CALL SET-UP

4 STRUCTURE OF IAS SOFTWARE

One of the most important aims of the IAS project has been the attempt to draw up a set of guidelines to assist in the division of a complete body of software among the processors in a multi-processor system, so as to gain the maximum advantage from the distributed control architecture. The guidelines which we finally deduced (and which are set out at the end of this section) are based on the suggestions made in the DISCUS operating system report[1], but these have been modified and extended, and some new ideas have been added, in the light of experience gained in using DISCUS for the IAS application.

It must be said at the outset that this software division is very dependent on the application concerned. Throughout this section we have tried to point out where the guidelines suggested result from particular properties of the IAS software, and how the division might have been done for different types of application. We have endeavoured to make the "list of rules" at the end of this section as general as possible, but inevitably the relative importance of these rules will be different for different types of application. Indeed, in applications very dissimilar to IAS, there may well turn out to be important points affecting the division of the software which are not considered in this report.

4.1 Definitions

Now, before proceeding further, a few definitions will be helpful. Most of those given below are also stated in the operating system report[1] in a slightly different form, but it will do no harm to repeat them here. Note that most of the definitions apply equally to co-operative scheduling and competitive scheduling schemes, but that IAS uses only co-operative scheduling.

(1) PROCESS

> A process is the doing of a particular job for a particular set of parameters.

Of course, something must define exactly what the job consists of, and in computing this "something" is a program. The program which tells a process what to do is called a function, and this leads to the definition:

(2) FUNCTION

> A function is the program which tells a process what to do under all possible circumstances.

It consists of a "closed" piece of code, by which we mean that there is no path through the function which involves a transfer of control to code outside of the function except to the operating system.

We are now in a position to say a little more about the relationship between a process and a function. A process consists of a set of values for the variables declared within its supporting function, which together with the values to be loaded in the processor's registers form the process context. Consequently a function may support any number of processes subject to the limitations imposed by the store available. Therefore the number of processes actually executing within any system is limited to the number of processors which the system has available to run them on. In a single processor system this is, not surprisingly, one, and the operating system scheduler has the job of deciding which of the available sets of parameters to pass control to next.

In a multiprocessor system, there may be more than one active process at any one time and these processes may even be active on the same function provided that several processors have access to the appropriate code.

(3) PROCESS TYPE

> All processes supported by a function are said to be of the same process type.

They all use the same supporting code and have the same form of context data. Processes which are supported by different functions are said to be of different process types. This may seem to be so self-evident as to be hardly worth mentioning, but for reasons of expediency it is sometimes convenient to amalgamate several functions into a single program, running on a single processor. This program then, of course, supports processes of several different types, and to avoid duplication of software, these different types of process may share some, but obviously not all, of their supporting code. Such composite programs are sometimes referred to as "functions", but the word does not carry the same meaning as in the previous definition. In this report we make the distinction between functions, as defined above, and DISCUS functions, which consist of one or more such functions. We can summarise this in the following definition:

(4) DISCUS FUNCTION

> A DISCUS function consists of one or more functions loaded onto a single DISCUS processor.

(5) FUNCTION DUPLICATION

> Function Duplication is a means of increasing processing power by loading a DISCUS function on to more than one processor, in order to increase the function's throughput.

(6) PROCESS STATE

> The state of a process at any time is defined by the values of all the variables in use by the process at that time.

These will a' clude the values stored as process context, and may also include the va undry "scratch pad" variables.

(7) WAIT POINT

In general, a running process will reach a state where it cannot continue without further input data. The process has then reached a wait point, and thus we can say:

> A wait point is a state of a process in which it requires more input data before it can continue executing.

Since IAS uses only co-operative scheduling, a process is always at a wait point when it is activated, and always suspends itself at a wait point. Simple processes may have only one wait point, in which case each process completes having received only one initial set of input data, and no context need be preserved. More complex processes may have many wait points, and may need to preserve relatively large amounts of context information. We will return to the subject of context preservation later, in section 4.2.2.

(8) TRANSACTION

> A transaction is a set of transient data which, when read, causes some processing to be done.

The processing which is done depends on what processing entity we consider to have read the data. For example, if a process reads the data, then the resultant processing is done by that process, and is complete at the next wait point. If, however, we are considering the entire system, then the processing which is done as a result of the transaction data being read is all of the processing done by all the processes in the system, until such time as it requires more data to allow it to continue.

4.2 Definition of Software in Terms of Processes

There are two different but inter-related aspects of a complete software job which need to be studied in order to decide on how the job should be divided into sub-jobs for which functions can be written. These are:

(1) To decide broadly what identifiably different jobs the software has to perform, that is, perform a functional decomposition to some desired level of detail.

(2) To determine how the software is constrained to run by the nature of the job it is trying to do, that is, establish the "run-time structure" or "process structure" of the application.

The term process structure can be explained as follows. Probably any large, real-time software package can be considered to consist, when running, of a number of processes, as per the definition. These processes may all be different, or may consist of one or more groups of similar processes, but the point to note here is that each process will be related in some definite way to the job which the software was written to do. For instance, in the case of setting up local calls in IAS, we might choose to have a process for each call in progress, or a process for each channel on which a subscriber terminal exists, or a process for dealing with each type of CPC which could be received. There are probably many more alternatives. In selecting one of these we need to consider:

(a) Whether a process structure reflects the way in which we think about the job to be done (since this makes it easier to understand), and whether the necessary context information, organised to suit a particular process structure, can be readily interpreted by the human programmers.

(b) Whether the application is such that the whole job can be divided into sub-jobs, at least some of which can be done simultaneously by different processors, and if so whether the process structure allows the maximum amount of parallel processing to be carried out, without the division into sub-jobs becoming so contrived as to make the software incomprehensible. Parallel processing and its impact on applications

software are discussed more fully in section 4.3.1.2.4.

Evidently it is helpful to know how the complete job can be conveniently divided into sub-jobs before we start to decide on a process structure, since it may be easier to invent process structures for sub-jobs than a complete job, particularly if the sub-jobs are fairly small, and can be considered largely independently for this purpose. Conversely, it is helpful to know the kind of process which each function will have to support before deciding on which sub-jobs, (or groups, or parts thereof) to consider as DISCUS functions. Thus functional decomposition and the devising of a process structure are both inter-related and iterative, and it is important that both aspects be considered right from the start of the project. (See Appendix A, "IAS Mark I - The Great Mistake" for an example of how not to do this.)

Given that we have decided what functions are required to support all the necessary processes, all that is needed is that each function be loaded, possibly with others, onto one or more DISCUS processors and the machine can start doing its job. The original way in which it was envisaged that DISCUS would be used was that the architecture of the DISCUS hardware would prevent one process from corrupting another's context. This was to have been ensured by keeping the context data for different processes in the local store of different DISCUS processors, which implies that there is one process per processor. It is at this point that the usual unfortunate divergence between ideal systems and physical reality becomes apparent. The process structure which seems most suited to the IAS application involves having one process per communications channel (see section 4.2.2). Since there are 5 groups, each having 30 traffic channels, this implies a total of 150 processes to handle the communications channels alone, to say nothing of any other types of process which may be required (again, see section 4.2.2). The DISCUS system at present available will support a maximum of 15 processors, with the possibility of this figure being raised to 24 without too much effort. It may or may not be technically feasible to produce a usable DISCUS of over 100 intercommunicating processors, but the cost would in any case be prohibitive. In dividing the IAS software among the available DISCUS processors we therefore compromised by running all processes having the same supporting function on the same processor. This uses DISCUS more efficiently, since it avoids the considerable duplication of code which would occur in the ideal case, but it does mean that the protection facilities provided by the DISCUS hardware are not fully utilised, since it is now physically possible for one process to corrupt the context of another similar process. Moreover, once more than one process can run on a processor, the function loaded on that processor must include some form of scheduler to decide which process is to be run, which complicates the application software somewhat. However, if all the context is held in global store, the DISCUS operating system provides some software protection, since it formalises access to the context data.

To summarise then, when dividing a software package into functions, the following points should be borne in mind:

(1) Functional decomposition, that is, the design of the static structure of an application, and the design of the process structure of the application are very interdependent.

(2) The process structure should be related to the jobs to be done in some obvious way.

(3) If the application is to run on a DISCUS system, and the hardware protection facilities provided by DISCUS are to be fully utilised to protect process context, the application must be such that a convenient process structure can be devised which requires no more than 24 processors.

(4) Even when we know what processes are needed for an application, and what functions are needed to support them, if there are more than 24 processes, additional factors (e.g. the number of processors available, the cost of the system per processor) must be considered before we are in a position to allocate functions to DISCUS processors, other than arbitrarily. In fact, it seems that the division of the applications software into functions, and the mapping of those functions onto DISCUS processors, are largely independent.

The next three sub-sections describe in more detail the design of the process structure and static structure of the IAS applications software. Many of the remarks in these sub-sections apply equally to a multi-processing system running on a single processor as well as a DISCUS multi-processor. Section 4.3 explores the mapping of the IAS software onto separate DISCUS processors, taking into account the general discussion that precedes it.

4.2.1 IAS Functional Decomposition

The IAS software can be considered to consist of three major parts:

(1) Call set up and clear down
(2) Data-base management
(3) Manual control and testing facilities

It does not necessarily follow that each of these parts will be coded as a DISCUS function. At this stage the functional decomposition merely gives an indication of the job which the software has to do.

4.2.1.1 Call Set Up and Clear Down

The job this software has to do is precisely what its name suggests. An outline of what is involved has already been given in section 2, and so no more need be said at this stage.

4.2.1.2 Data-base Management

The job may also be sub-divided into three parts:

(1) Subscriber connectivity management. It is necessary for the IAS data-base to show the positions of all local subscribers, that is, those connected directly to this node. Since subscribers in a tactical area network can move from one node to another, some means must be provided by which a subscriber can signal to his parent node his impending departure, and indicate his arrival at a new node.

Subscriber connectivity is also affected by the use of the "call transfer" facility, and the need to contain information on the position of certain non-local subscribers (the Frequently Called List or FCL). When the IAS has to set up a call to a non-local subscriber whose position is not in the data-base, it transmits a search message to its neighbouring nodes, and thence throughout the network, to locate the wanted subscriber. This is known as a Flood Search, and is the responsibility of the data-base management software in IAS.

(2) Node connectivity management. The data-base of each node in a tactical area network must include the inter-node connectivity of the network. This allows a node which has to set up a trunk call to determine which of its trunk groups to use as the first link in a route leading to the node to which the wanted subscriber is connected. A node can obtain updates for its connectivity map from neighbouring nodes by some suitable exchange of messages. The IAS data-base management software must handle the getting of this information, its inclusion in the data-base, and the forwarding of such data to other neighbouring nodes.

(3) Archiving. To facilitate recovery with minimal loss of information in the event of a serious software failure, certain parts of the IAS data-base are copied to backing store (a floppy disc) at regular intervals.

4.2.1.3 Manual Control and Testing

This software provides the IAS man/machine interface, which is used mainly for debugging and testing applications programs. The development aids provided are:

(1) The facility to send messages to any DISCUS function in the system. This allows processes to be stimulated manually so that functions can, if necessary, be tested one at a time. The messages are entered at the system console.

(2) A data-base editor which allows data in global store to be displayed on the system console, and manually altered.

(3) The facility to receive fault messages from any DISCUS function in the system, and display them on the system console.

The functional decomposition, as described so far, is shown in fig 4.1.

4.2.2 IAS Process Structure

For simplicity we initially concern ourselves only with the setting up and clearing down of local calls.

A compelled signalling system, such as that implemented in IAS, implies that for every code sent by a subscriber terminal to the switch, the switch must send a code to the terminal in response. This requirement for compelled responses determines one of the fundamental run-time characteristics of the IAS software, namely that processing must be performed in relatively short bursts between receiving input data and outputting response data. The processing to be done in each burst will depend on the particular code received as well as the history of previous code reception, so that several different sections of software are required, one for each different code. Thus the processing performed by the IAS software consists of a sequence of operations, each of which can be represented as:

WAIT -> INPUT -> PROCESS -> OUTPUT -> WAIT

Since the processing to be performed when a code is received depends in part on the previous history, some means of storing this historical information (the context) must be provided. This becomes more complicated when several subscribers are all setting up and clearing down calls concurrently, as is generally the case. Indeed, in IAS, up to 90 subscribers on different channels may all be active simultaneously, and it is necessary to store context information relating to all the processing being carried out in the switch. What

this context information consists of, and the way in which it can be most conveniently organised into data objects, depends on the job which the whole software package is trying to do. It is the organisation of the context information which determines the process structure of the software.

In IAS, for local call working, the context information required is a history of the codes which have been received, and the channels on which they were received. This history need not, of course, be complete, (we do not need to store every code which has ever been received on each channel, since once RELEASE has been received, only SEIZE is expected from that channel, irrespective of past history) but it must be sufficient to allow the switch to carry out its job.

There is evidently a very large number of ways in which the context information can be organised, but it seems sensible to arrange it in a way which reflects the setting up and clearing down of telephone calls, as understood by human programmers. We think of tactical area network telephone call processing in terms of "codes", "channels" and "calls" and indeed the IAS hardware deals with data as group/channel/code triads. If we arrange the context as a sufficiently large number of these triads, there is still a very large (probably infinite) number of ways in which we can use the information to set up calls. However, most of these would be highly obscure, and a few of the more easily understood, and therefore more attractive, options are set out below.

Whatever context organisation we choose, we need to devise some software which can perform all the processing which can be needed when any code is received. When a code is received, this software will need:

(a) The code itself
(b) The group and channel on which it was received
(c) The relevant history of codes previously received

Given all these, a process can run on the piece of software. There are several ways in which we can describe the process structure of this software:

(1) We can consider the processes to be channel based. That is, we have one process per active channel. Such a process is deemed to be started when a certain code is received from (or in the case of a called channel, needs to be transmitted to) a subscriber terminal. When all the processing which can be done with the information available has been completed, the context information relating to the channel concerned has been altered, and the process is suspended. It is then awaiting more input data, i.e. another transaction. Eventually a transaction will occur which causes the context information relating to the channel to be indistinguishable from its state immediately before the process was started. When this transaction has been processed the process is said to have been stopped, i.e. it is in an idle state. As far as the software is concerned, the choice of this particular state of the context from all the various states it can assume is arbitrary. However, in human terms it seems helpful to choose as this quiescent context state that which represents the channel idling, i.e. when the subscriber terminal connected to the channel is on hook. In this state the amount of context to be stored will be a minimum.

Evidently some software is required to decide which process to stimulate into action, by determining on which channel a code has been received.

(2) We can consider processes to be code based. That is, we have one process per code (one for SEIZE, one for RELEASE, and so on). Such a process is started when the code to which it is capable of reacting is first received, on any channel. When all the processing which can be done with the available information has been completed, the context relating to the code concerned has been altered and the process is suspended. It then awaits another transaction. It is unlikely that the context relating to a code will often become indistinguishable from its state before the code was first received, since the codes received on different channels are almost independent. Even if the context were ever to return to this state, it would not necessarily be significant in terms of human understanding of making telephone calls. Code based processes are probably best considered to run indefinitely. (Channel based processes could, of course, be considered like this too.)

Evidently some software is required to decide which process to stimulate into action, by determining which code has been received.

The above two process structures are very similar. The only difference is that in (1) a code is passed to a channel based process as a transaction, and in (2) a channel is passed to a code based process as a transaction. Of the two, we think (1) is closer to the normal way of thinking about the setting up of telephone calls.

(3) We can consider processes to be call based. This is somewhat different to the previous ways of thinking about process structure since a call is a more abstract concept than a channel or a code. The latter two can have numerical values, which are received from and sent to the IAS communications hardware. The notion of a call arises out of the way humans think about how telephones are used. Call based processes would have to be assigned dynamically to calls, since a call has no identity with which a process context can be marked until the call is actually in existence. Some additional software would therefore be required to perform process allocation. Furthermore, since the IAS hardware deals with input and output data on a per channel basis, there must be some means of relating this data to the call concerned. This channel to call mapping would have to be done for every received code, and the reverse mapping performed for each piece of output data, which seems rather wasteful of processing power.

From the above, a channel based process structure seems to be the most easily understood and easily implemented system. This type of structure was chosen for the IAS local call processing software for ease of understanding of what is a predominantly channel based process structure.

The IAS trunk call processing software also has a channel based process structure, but since the stimuli received by the switch, the processing it must perform, and the responses it must make to set up a trunk call are different from those associated with a local call, different software is required to support the trunk call processing processes.

Some of the sub-jobs which the IAS software has to perform are not directly related to any particular channel, even though the stimulus which resulted in the sub-job being done arrived on a specific channel. Such sub-jobs are often logically separate from processing to handle stimuli received on channels, or can be carried out at the same time as this processing. An example of the logically separate type of sub-job is the making of a flood search. Such a search is carried out when a local subscriber has originated a call to a subscriber the identity of whose parent node is unknown. The parent node of the calling subscriber sends search messages to all neighbouring nodes, and these nodes, if they do not own the called subscriber, relay the message to

all their neighbours, and so on, until the parent node of the called subscriber is found. This node then sends back a reply to the originator of the search. When this reply has been received, a trunk call can be set up in the usual way. In IAS, flood searches are implemented using non-channel based flood search processes. This is done because:

(a) The making of a flood search cannot be included in a trunk channel process, since at the time when a flood search is originated, no trunk channel process has been assigned to the call which started the flood search, because the group on which the call is to be set up is unknown.

(b) If the calling subscriber goes on hook before the reply has been received, it is useful if the search continues, and the position of the called subscriber in the network established, because the calling subscriber will probably make another attempt within a short time. If the making of the flood search were part of the process for the calling (local) channel, then the flood search reply would no longer be expected and would be ignored.

Call transfers are implemented by non-channel based processes for similar reasons. Several processes of each type will be needed so that more than one flood search or transfer implementation can be carried out simultaneously. Further, (different) processes will be needed to deal with flood searches and call transfer requests received from other nodes.

An example of the type of sub-job which, though the initial stimulus for it is received on a particular channel, can be done simultaneously with other processing on that channel, is archiving of context information in backing store. This is done by a separate, non-channel based archiving process.

Other sub-jobs are not related at all to any particular channel. These include node connectivity management, and the manual control and testing facilities. These too use non-channel based processes.

To summarise, the process structure of the main part of the IAS software, i.e. that part that deals directly with call handling, is channel based. However, for the reasons outlined above, it is convenient, and in some cases essential, to use some non-channel based processes to carry out certain jobs not directly related to call management. Thus the need for the following types of process has been identified:

(a) Call management processes:
    One for each local channel
    One for each trunk channel

(b) Data-base management processes:
    Archiving context information
    One for each of a number of flood searches
    Respond to flood searches from other nodes
    One for each of a number of call transfer implementations
    Respond to call transfer requests from other nodes
    Send out connectivity updates
    Handle connectivity updates from other nodes

(c) Manual control and testing processes:
    Receive and display messages on the system console
    Send messages to other processes
    Data-base editing

4.2.3 IAS Static Structure

In section 4.2.2 we identified the need for several different types of process. All processes of the same type (i.e. having similar blocks of context) can run on the same code. Thus, to support the required processes, the following programs must be written:

Local channel program
Trunk channel program
Archiver
Flood search initiator
Flood search responder
Call transfer implementor
Call transfer request handler
Node connectivity program
Message display program
Message sender
Data-base editor

In addition, peripheral device handlers will be required for:

Code Detector Unit (CDU)
Trunk Signalling Unit (TSU)
Digital Switching Matrix
Real-Time Clock (RTC)
Floppy Discs
Console

4.3 Software Mapping onto DISCUS

As was mentioned at the beginning of section 4.2, it was originally envisaged that only one applications process would be run on each DISCUS processor. In the case of IAS, this proved impracticable because of the number of processes required, and several processes had to be run on each processor, thus sacrificing some of the protection against accidental context corruption provided by the DISCUS architecture. However, by placing all context information in global store, and accessing it only via the operating system, a considerable measure of software protection can be obtained.

If an application were to use a DISCUS system on a one process per processor basis, it is probable that considerable processing time would be wasted, even though the process context is protected to the maximum extent. This is because each processor can only do one thing, and may therefore have to spend a large proportion of its time waiting for a transaction it can deal with to arrive at its input. This inefficiency may not matter in some applications, but in other cases a different approach is required.

In cases where the whole of the applications code will fit into the address space of a single processor, the greatest efficiency in the use of processing time can be achieved by running all the applications processes on each DISCUS processor. Thus every processor in the system can do the whole job for which the applications software was designed, without having to wait for any other processor (except when access to global data is required). Indeed, a minimum system in this case would consist of only one processor. We have, of course, sacrificed the hardware protection of the DISCUS architecture, but if it has been necessary to depart from the one process per processor approach for other reasons (as with IAS), this protection has already been lost. In any case, in a configuration such as this, where more than one processor is supporting the same function, we are forced to place all context in global store, since there

is no way of predicting which of the processors will deal with a particular transaction. If the context is then only accessed via the DISCUS operating system, a reasonable amount of software protection will be provided.

Between these two extremes (i.e. one process per processor. and all processes on one processor), there lies a range of possible ways in which an application can be mapped onto a DISCUS system. The IAS software, since it has too many processes for the first option, and too much code for the second, must use one of the intermediate configurations. The operating system report[1], makes certain recommendations about how this mapping should be performed so as to obtain the maximum benefit from the multi-processor nature of DISCUS. These recommendations recognise the existence of different types of application, but do not consider any application in detail. They are repeated and expanded below, with particular reference to IAS. Inevitably the way in which the IAS software was distributed among the DISCUS processors was a compromise between the recommended ideals and various other considerations (e.g. the number of processors available). The way in which we arrived at this compromise is described in section 4.3.2.

4.3.1 DISCUS Recommended Practices

Nothing which has been said so far in section 4.3 takes any account of the problem of input and output in a DISCUS system. The operating system provides no I/O facilities, so that all device handlers must be written entirely as applications code. IAS has a fairly large number of peripheral devices, so that in this application I/O is quite important. For this reason the recommended practices are discussed below in two parts: those for DISCUS functions supporting processes which have to perform I/O operations and those which do not.

4.3.1.1 DISCUS Functions Supporting Peripheral Handling Processes

Perhaps the most important point to note about peripheral handling functions in DISCUS is that, unlike other DISCUS functions, they cannot be duplicated. That is, without an inordinately complicated hardware interface, it is not possible for two or more DISCUS processors to access the same device. Therefore, we must make such functions execute as quickly as possible so as to reduce the probability of their becoming bottlenecks. This implies that such functions should do as little processing as possible, and that as much of the applications code as possible should be implemented in functions not involving I/O, so that these functions can be duplicated if necessary. Such small, fast I/O handling functions have been called Device Handlers in IAS.

Ideally, Device Handlers should handle either input or output, but not both, i.e. they should be uni-directional. There are several reasons for this:

(a) A uni-directional device handler will input or output more rapidly than a bi-directional one, and is thus less likely to become a bottleneck.

(b) A uni-directional device handler can be written to have only one wait point: the point at which it waits for data from the peripheral device in the case of an input handler, or from a DISCUS channel in the case of an output handler. This simplifies the job of writing and testing the function, and facilitates recovery in the event of a crash (see the operating system report[1] sections 5.3.3 and 5.3.4)

(c) Uni-directional device handlers help to avoid the channel loop configuration in which deadlock can occur (see the operating system report section 5.4 and section 4.3.1.2.5 of this document for more detail). Such a looped configuration of DISCUS functions will occur if a bi-directional device handler both sends data to and receives data from the same other DISCUS function.

However, an output device handler may have to receive a status word from the device it controls, and an input handler may have to send a command word to its device. Any attempt to separate these operations from the actual input or output operations results in more than one processor being connected to the peripheral device (which is wasteful of processors and makes the hardware interfacing very complicated), or in the interpretation of the control words being done by a non-device handling function (which has to be connected to the device handler by two channels instead of one, allowing the possibility of channel deadlock, and also gives rise to problems if the non-device handling function has to be duplicated). For these reasons, device handlers must handle their own status and command words.

In almost any computing system there is a need to convert data received from the outside world into a form more suitable for use within the computer system than that in which it arrived, and the reverse conversion must be performed when an output is made. Strictly speaking it is not necessary for DISCUS device handlers to perform such conversions, and data should be passed to a non-device handling process running on a different DISCUS function as it is received, one byte at a time. However, when data is received from peripherals as a set of bytes, this would be a very inefficient way to use a DISCUS channel, and provided the assembly of the received data into a message can be accomplished quickly (as, for instance, in the case of the group/channel/code triads received from the CDU, or even trunk messages from the TSU), it is useful if this is done within a device handler. However, complex format conversion, or bit packing and unpacking will in general be too time consuming, and are best left to non-device handling translator processes.

4.3.1.2 DISCUS Functions Supporting Only Non-peripheral Handling Processes

The main points to be borne in mind concerning the distribution of the non-peripheral handling part of an application among DISCUS processors are as follows:

4.3.1.2.1 Number of Processors in the Minimum System

One of the most important constraints on the way in which we map applications software on to DISCUS is the number of processors we can afford to have in the minimum system (i.e. a fully implemented system with none of the functions duplicated). In its present state of development, DISCUS can support a maximum of 15 processors. If we observe all the codes of practice for device handling functions set out in 4.3.1.1, eleven of the processors would be device handlers:

CDU handler
Matrix handler
Real-time clock handler
TSU group 3 input handler
TSU group 3 output handler
TSU group 4 input handler
TSU group 4 output handler
Console input handler
Console output handler

Disc input handler
Disc output handler

This is evidently impracticable, since it leaves very few processors for non-device handling DISCUS functions, and virtually no capability for function duplication. The number of device handlers was therefore reduced to 7 by allowing bi-directional peripherals to be handled by a single DISCUS function, leaving 8 processors to run all the non-peripheral handling processes listed at the end of section 4.2.2. As can be seen from this list, there are 12 different types of process, two types concerned with call processing, three types with manual control and testing, and the remainder with data-base management. There are too many different types of process to allow each type of process to be run on a separate processor, so at least one DISCUS function will have to support more than one type of process. The way in which the different types of process can best be grouped together depends on the points discussed below, but it seems obvious that any type of process which is likely to be busier than most should have a DISCUS function to itself. Such a function can be duplicated to avoid bottlenecks without also duplicating code which does not support the process type causing the bottleneck, though this is not too serious a problem since computer store is relatively cheap. It was assumed, (and this assumption has been justified by experience) that local and trunk call processing processes would be the busiest process types, and these were therefore given their own DISCUS functions. Separate DISCUS functions for these two process types were required since the total code required to support both types would not fit into the local address space of a single processor. However, the idea of having only one process type per DISCUS function seems generally attractive from the point of view of software simplicity, and the fact that this cannot be achieved indicates, that for IAS at least, a DISCUS system which could support many more processors would be more suitable than the present system.

4.3.1.2.2 Parallel Processing

In order to obtain the maximum throughput and minimum response time from an application on DISCUS, sub-jobs which can be done in parallel should be identified, and done by separate processors simultaneously. Consider the three main sub-divisions of the IAS software as stated in section 4.2.1:

Call management
Data-base management
Manual control and testing

In the setting up and clearing down of a single call, either local or trunk, there is almost no scope for parallel processing, since the whole operation is essentially sequential. However, the processing of a call on one channel is usually independent of the processing of a call on another, so that the channel based call processing processes can profitably be run on separate processors. Given that all local channel processes are supported by the same DISCUS function, and all trunk channel processes are supported by another function, the only way to achieve simultaneous parallel processing on different local channels (or trunk channels) is by function duplication.

All the data-base management processes listed at the end of section 4.2.2 are largely independent of each other, and throughput and response time could therefore be optimized by running them on separate processors. However, there are insufficient processors to allow each type of data-base management process to have a DISCUS function to itself, so that we must use DISCUS functions supporting more than one process type. Other functions (see below) may influence this grouping of process types.

Similarly, all the manual control and testing processes are independent of each other, and of the data-base management processes, so that they too could, with advantage, run on separate processors. Since these processes are unlikely to be very busy (two of the three types being dependent on keyboard input) the efficiency advantage to be gained is small.

4.3.1.2.3 Size of Functions

A DISCUS function must obviously be sufficiently small to fit into the local address space of a DISCUS processor. As DISCUS and its system generator[1] are currently configured, this means that a DISCUS function can occupy up to 32 kbytes. Both the call processing functions are within this limit, though in the case of local call processing there is not much room to spare. It seems reasonable to suppose that at least one DISCUS function will be required to support the manual control and testing processes, and at least another one to support the data-base management processes. Bearing in mind that the use of 7 device handler functions has been proposed, and that local and trunk call processing will require at least one processor each, six more processors are left to support the remaining process types. The code on which the manual control and testing processes run can easily be fitted into 32 kbytes. The whole data-base management package is, however, unlikely to fit into a single processor's local address space, so that at least two processors supporting two different data-base management functions will be required. This occupies 12 processors, with only 3 remaining for function duplication. Therefore, if we wish to conduct any experiments to determine the effects of function duplication on system performance using the present DISCUS system, the distribution of the IAS software among the processors will probably have to be done at least partly in contravention of the recommended practices previously described.

Because it is convenient to allocate applications software design tasks on the basis of one function per person, functions should be sufficiently small that each can be understood thoroughly and in detail by one person. In writing the IAS functions, it has been found that software packages of up the the system imposed limit of 32 kbytes can be so understood without too much difficulty.

4.3.1.2.4 Run-time Efficiency

The inefficiency inherent in the one process per processor approach was discussed at the beginning of section 4.3. Having been forced to abandon this approach in the case of IAS because of the large number of processes required, we find that running more than one process on the same processor increases efficiency, but at a cost of reduced hardware context protection, and increased software complexity, because of the need for a DISCUS function supporting more than one process type to include a scheduler to decide which process should run.

If such DISCUS functions are used, the efficiency of the whole system is very dependent on the way in which the sections of code supporting the different processes are grouped together to form the DISCUS functions. It will also depend on the process structure (as described in section 4.2.2) and on the way in which the DISCUS functions communicate with each other, as described below.

In considering run-time efficiency, it is important to define exactly what it is that the system is supposed to do efficiently. If we require a high throughput, then this can be achieved by dividing the job to be done into several sub-jobs which can be done one after the other, and implementing these sub-jobs as functions running on separate processors connected in series by DISCUS channels as shown in fig 4.2b. For example, if a system has to handle a

succession of similar jobs called 1, 2, 3 etc, each consisting of sub-jobs A, B and C, then a single processor must complete all sub-jobs for job 1 before it can start job 2. However, if A, B and C are done by three separate processors in series, then the first processor can do sub-job A for job 2 while the second processor is doing sub-job B for job 1, and so on. Note though, that this series configuration of processors gives no improvement in response time over that achievable with a single processor (fig 4.2a), unless the speed of response is limited by the existence of an input queue, which will be reduced by the use of several processors in series. Indeed, response time may increase because of the time spent in passing data from one processor to another. There is also the possibility that one sub-job may take substantially longer than the others, so that a queue will build up at the input to the DISCUS function concerned and the other processors downstream will waste time waiting for input data. If an improvement in processor limited (as opposed to queue limited) response time is required, then either the code itself must be made more efficient, or some sub-jobs must be done in parallel. If parallel processing is possible, then an increase in throughput will also occur, since if each complete job can be done more quickly, more jobs can be done in a given time, provided that the function, X, which sorts out the sub-jobs is efficient (see fig 4.2c).

In applications where parallel processing on the same job is not possible, as in the case of a single call in IAS, or impossible to identify, no improvement in processor limited response time is possible by this means. However, duplication of a single function, which allows parallel processing on different jobs, is a better way of increasing throughput than using several smaller DISCUS functions in series, because all processors can then do all sub-jobs, so no time is wasted due to bottlenecks (see fig 4.2d). There is, however, no point in providing more copies of a function than the maximum number of jobs which can be present simultaneously. For instance, in IAS there is no point in providing more copies of local call processing than there are channels on which calls can be made.

In applications where it is not possible to fit all the code required to do the whole job into one processor's local address space and parallel processing on the same job is not possible, we may be forced to adopt the serial approach. The number of series processors needed to give the required throughput and response time will depend on the rate of arrival of new jobs, the overheads involved in passing data from one processor to another, and the number of processors allocated to device handling functions. If the application is such that parallel processing on the same job is to some extent possible, there is no reason why the series and parallel configurations should not be combined to improve processor limited response time. If parallel processing on the same job is not possible, it may be worth considering the duplication of one or more DISCUS functions in the series in order to improve throughput, particularly if message passing overheads are high, see fig 4.2e.

For maximum efficiency it is better to keep inter-function communication to the minimum possible, consistent with the matters discussed above. This minimizes the time each processor spends waiting for the use of the global bus, though in practice no problem has arisen from lack of global bus capacity. More important, minimising the extent to which global arrays are accessed by more than one processor also reduces the amount of time one processor spends waiting for another to release an array which it wants, and also reduces the chance of deadlock occurring.

4.3.1.2.5 Functional Connectivity

Certain configurations of DISCUS functions and interconnecting channels should be avoided. The first of these is the string of functions in series (as in fig 4.2B) which, as described above, is inefficient because of the time spent in passing messages from one function to another, and because of the possible occurrence of bottlenecks.

Another undesirable configuration is any which includes a closed loop. The simplest example of this type of arrangement is shown in fig 4.3. The use of such configurations can lead to deadlock. Consider two DISCUS functions, f1 and f2, connected by two channels, c1 and c2. If while f1 is putting a message into c1, the channel becomes full, f1 will have to wait until f2 removes some data from the channel before it can finish putting in its message. If, at the same time, f2 is putting in a message to c2, and this channel also becomes full, then f2 must wait until f1 removes data from c2. This f1 cannot do, because it is waiting on c1, and the two functions are deadlocked. Even if f1 and f2 received no further transactions from elsewhere, the deadlock will continue, and at least one of the processors must be reset in order to restore the system to correct operation. This situation is particularly likely to arise if a bi-directional device handler both takes data from and gives data to the same other function, but loops involving more than two functions should also be avoided.

Having said all this, it must be noted that with an application as complex as IAS, which needs to use several bi-directional peripherals, running on a hardware system which can support only 15 processors, it is virtually impossible to avoid channel loops. In fact, DISCUS functions in IAS are almost fully connected, and the system therefore contains a large number of inter-connected channel loops. However, the probability of channel deadlock occurring can be reduced by making the channel buffers as long as possible, so as to reduce the chance of any buffer becoming completely full.

One further point about functional connectivity is worth noting. If two different DISCUS functions both produce data which needs the same processing before being passed on to a third, code duplication can be avoided by placing the code to process both lots of data either in the receiving function, or a separate function. Of course, we may wish to duplicate this code to increase throughput, but in that case the code to be duplicated would be better arranged as a separate function, which could be duplicated if necessary. The alternative is to have two functions which, in part, perform the same processing, and are possibly written by two different programmers, in which case it is likely that the same processing will be done in two different ways. At best this implies inefficient use of the programmers. At worst, the output data produced by the two functions, although nominally in the same form, may be subtly different, and the run-time consequences may well be unpleasant.

4.3.2 Configuration of DISCUS Functions in IAS

The main constraint on the mapping of the IAS applications software onto DISCUS has been the small number of processors available (up to 15) and the large number of IAS peripherals, and therefore device handlers. In the early stages of the project, the shortage of available DISCUS hardware was an even greater problem, and forced a drastic reduction in the number of device handler processes which could be used.

Initially, in order to get some sort of system running, only local call processing was attempted. Thus the only peripherals required were the matrix, the CDU and the real-time clock. The device handling code for these was combined into a single DISCUS function, so that the minimum system consisted

of only two processors, one supporting the device handlers and one the local call processing processes. In order to load the software (which was at this stage subject to frequent modification) conveniently, the executable code was taken from a floppy disc and sent to the processors concerned via the DISCUS global bus and store. This, of course, implies the need for a disc handler. Because a hardware interface between the floppy discs and the Intel MDS 800 development system already existed, the MDS was coupled to the DISCUS global bus and used as a loader as well as disc handler. Once the MDS was connected to the global bus it could, of course, be used as a DISCUS processor. The most obvious program to load on to it was the IAS disc handler, but because of the shortage of DISCUS processors at the time, it was considered that rather greater use should be made of the MDS in this capacity. Since the IAS disc handler will be used predominantly for archiving data, and because archiving comes under the general heading of data-base management in IAS, it was decided that initially the MDS should support all the data-base management processes implemented. This decision is evidently in conflict with all that has been said previously about keeping device handlers small and efficient, but it sufficed to get things running and could be revised later.

To facilitate the testing and debugging of the IAS software, it was found essential to be able to stimulate processes with messages from the console, to display received messages from processes on the console, and to manipulate the data in global store. Therefore, another processor was required to support the manual control and testing processes. Supporting all three of these processes with one DISCUS function, and including in the function the console handlers, seems a reasonable step to take, since none of the processes involved are likely to be very busy.

Thus the first implementation of IAS on DISCUS consisted of four processors connected as shown in fig 4.4. Each arrowed line joining two DISCUS functions in this diagram represents a channel. Representation of channels in the normal Activity-Channel-Pool (ACP) format is cumbersome where large numbers of channels are involved. Notice that there are several channel loops in this configuration. Most of these arise from the need for the manual control process running on the General Purpose eXerciser, (or GPX), to send messages to other processes and receive messages from them. These loops are unlikely to give rise to channel deadlock because of the slow rate at which GPX sends out messages, these being originated at a VDU keyboard. The call processing/device handler loop, however, is more likely to deadlock.

When the IAS, as configured in fig 4.4, was tested using a traffic generator to emulate a large number of subscribers making calls (see section 6), it was found that the largest queue of messages formed in the input channel to the local call processing function (CLP) at higher calling rates. This indicated that CLP was forming a bottleneck in the system. Duplicating CLP relieved this bottleneck, but triplicating the function gave little further improvement in throughput, because the largest queue now formed in the input channel of the Device Handler (DH). This function could not be duplicated, of course, so that the only way to obtain an improvement in throughput was to arrange for the matrix, CDU and RTC to be handled by separate DISCUS functions, as recommended previously. The configuration of these functions was then as shown in fig 4.5.

The next stage of development will be to add a function to support trunk call processing (CLPT) to the system, together with separate handlers running on separate processors for each of the TSUs (TSUH3 and TSUH4, to handle the TSUs on groups 3 and 4 respectively). The conversion of trunk messages from the format in which they are received from the TSU to a form more convenient for use by the IAS software, (and the reverse conversion) are considered to be too lengthy to include in the TSU handlers, and a separate DISCUS function to perform translation will therefore be used. A bi-directional translator is contemplated initially. At the time of writing it is not known whether this

will cause a bottleneck, and require splitting into two uni-directional parts. If this is needed, a considerable amount of software will be common to both translator functions, which will nevertheless be different. Under these circumstances, the problems pointed out in section 4.3.1.2.5 can arise, especially if two uni-directional translator functions had been written in the first place. The configuration of the minimum system in this enhanced state is shown in fig 4.6. Note that this does not include all the data-base management software, in that processes concerned with inter-node connectivity have not yet been implemented. These will probably require an additional DISCUS function to support them, since all the data-base management software is unlikely to fit into a single processor.

The system, as configured in fig 4.6, is not in accordance with all the recommended practices set out in section 4.3.1. In particular:

(a) There are a number of channel loops, though the probability of their giving rise to channel deadlock is minimized by the use of long channel buffers.

(b) Not all DISCUS functions which handle peripheral devices do only that. The data-base manager (DBM) and GPX are offenders in this respect. However, in order to keep enough processors in reserve for function duplication experiments, we are forced to tolerate this.

(c) Bi-directional device handlers are used, which gives rise to channel loops. Because of the complexity of the hardware interface required to connect two processors to each bi-directional peripheral device, it does not seem worthwhile to change to uni-directional device handlers. In any case, the present DISCUS system cannot support enough processors to allow this to be done.

In summary, the practices recommended in section 4.3.1 were adhered to as far as this was conveniently possible. The main cause of departures from them were the large number of peripheral devices to be handled, and the relatively small number of processors which can be supported by the present DISCUS hardware.

4.4 Guidelines for Applications Software Structure

The following section summarises the preceding discussion as a set of rules to help in designing the structure of the application software. These are divided into two lists, one of which contains rules which apply to any multi-processor system, and the other rules which apply specifically to DISCUS. Notice that some of the guidelines contradict each other. This is frequently the case in attempting to formulate design rules, and it is necessary to trade off one advantage against another, giving greater weight to whatever happens to be more important in a particular application.

4.4.1 General Guidelines

(1) Both the process structure and the static structure of the application must be considered right from the start. This implies that functional decomposition and the devising of a process structure be done iteratively.

(2) If a "one process per processor" approach is desirable, the application must consist of no more than processes than there are processors to run them on.

(3) The process structure of the software should be related to the jobs the software has to do in some obvious way. It should also take account of the form in which data is received from, and sent to, the outside world.

(4) Division of the application into convenient functions, and the allocation of those functions to processors, are largely independent.

(5) Manual control and testing facilities should be designed into the system from the start.

(6) If all the applications code will not fit on one processor, we must use several. If parallel processing on the same job is possible, the use of several different functions in parallel will give high throughput and low response time.

(7) If parallel processing on the same job is not possible, and all the code will not fit on one processor, we must use two or more functions in series. This will give high throughput, but poor response time. Function duplication will improve throughput further, but will not improve processor limited response time. There is, however, no point in providing more copies of a function than the maximum number of transactions of the type it can process which can be in the system simultaneously.

(8) To use the available processing power efficiently, long strings of functions in series should be avoided where possible, because of the message passing overheads involved, and the probability of bottlenecks occurring.

(9) For efficiency, inter-function communication, both by channels and via global arrays, is best kept to a minimum. This also reduces the chance of deadlock occurring.

(10) Duplication of the same job in two or more different functions should be avoided.

(11) Functions should be kept sufficiently small to be understood thoroughly and in detail by one person.

(12) Device handlers should be uni-directional, to avoid scheduling within a function.

(13) Device handlers should handle their own peripheral command and status processing.

(14) Some data conversion in device handlers is allowable, provided it can be done efficiently and quickly. If not, the conversion should be done elsewhere.

4.4.2 DISCUS Specific Guidelines

(1) If more than one process is run on each processor, all process context must be kept in global store, since if a function is duplicated it is not possible to predict which processor will handle a given transaction. The operating system can provide some protection of the context from corruption. This is necessary, since the DISCUS hardware could not prevent processes running on the same processor from corrupting each other's context if this were in local store.

(2) The "one process per processor" approach utilises the DISCUS hardware protection mechanism to the maximum possible extent, but is inefficient. The most efficient approach is to place all the applications code, except device handlers, in a single processor, assuming it will fit, and then duplicate that function as required.

(3) Closed loops of DISCUS functions and channels should be avoided because of the possibility of channel deadlock. Where they cannot be avoided, the probability of deadlock occurring can be reduced by using long channel buffers.

(4) Device handlers should be small and efficient, and perform only device handling.

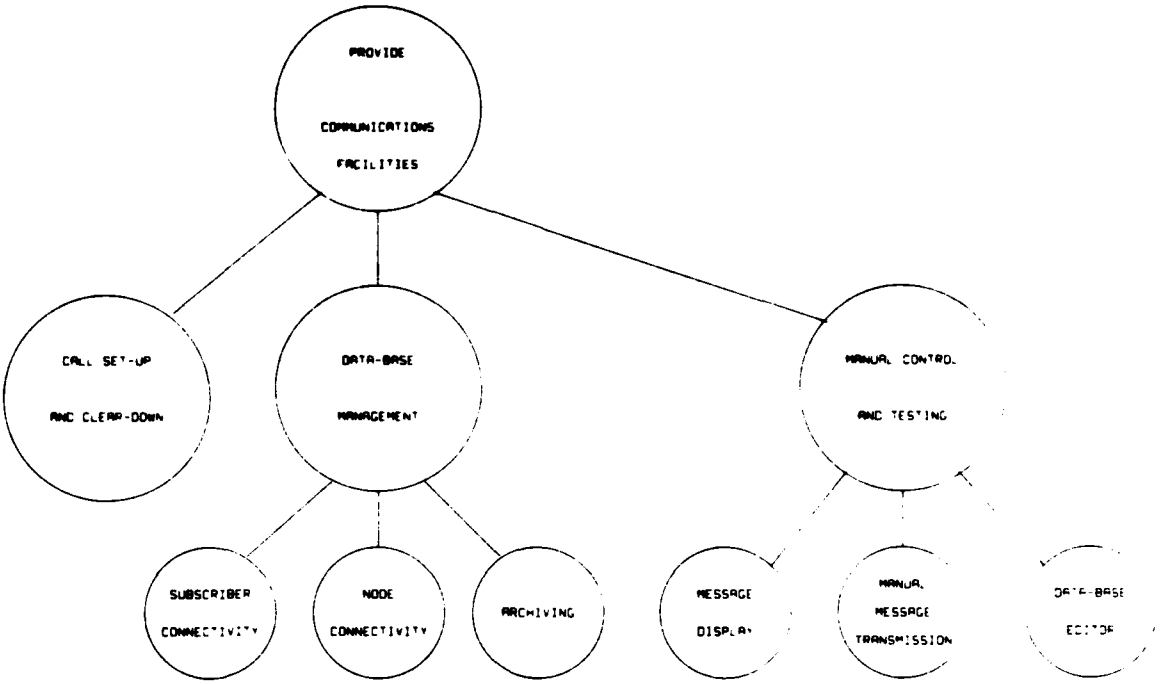


FIG 4 1 FUNCTIONAL DECOMPOSITION FOR IAE APPLICATIONS SOFTWARE

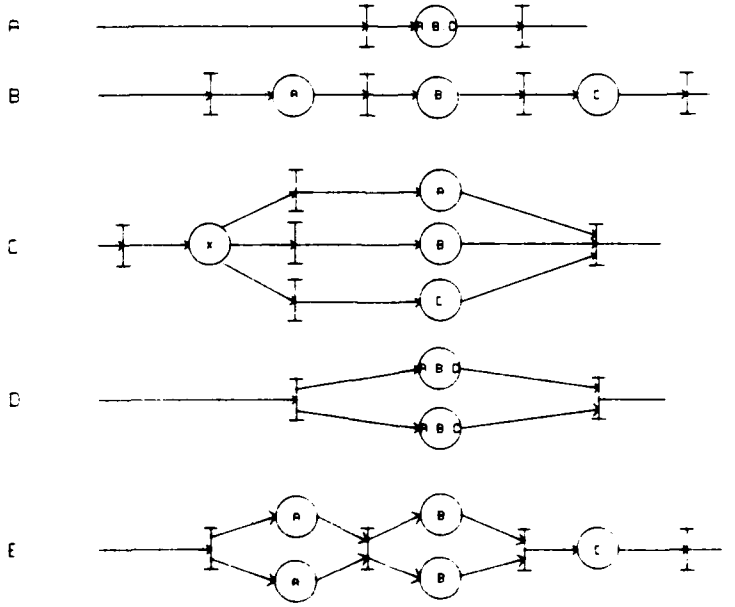


FIG 4 2 CONFIGURATION OF DISCUS FUNCTIONS FOR DIFFERENT TYPES OF APPLICATION

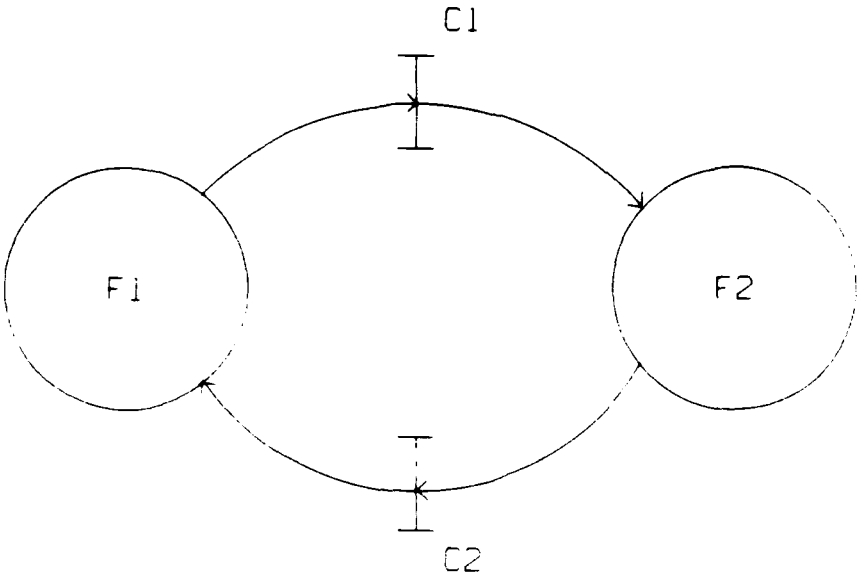


FIG 4 3 ILLUSTRATION OF CONFIGURATION IN WHICH CHANNEL DEADLOCK CAN OCCUR

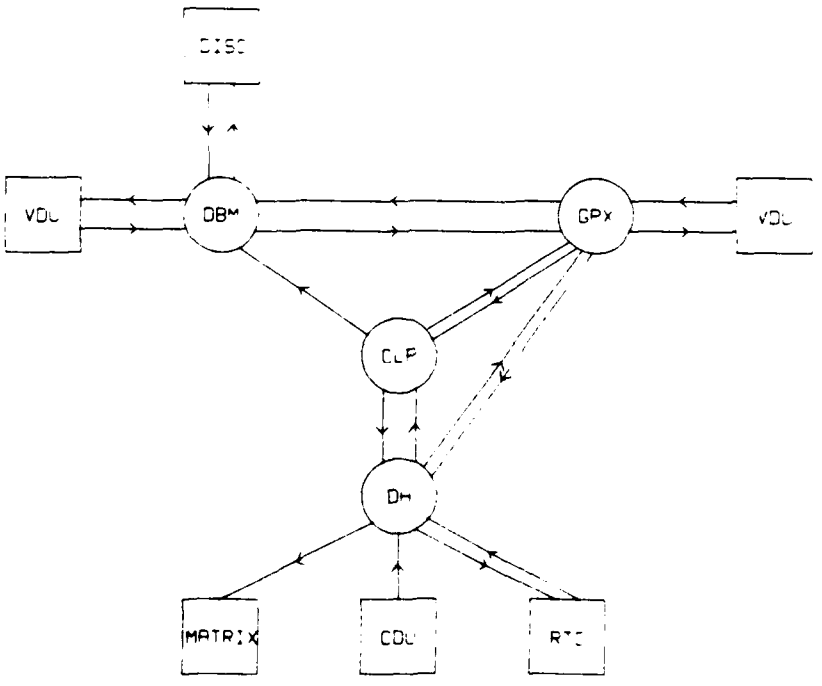


FIG 4 4 IAS FIRST IMPLEMENTATION

LOCAL CALLS ONLY

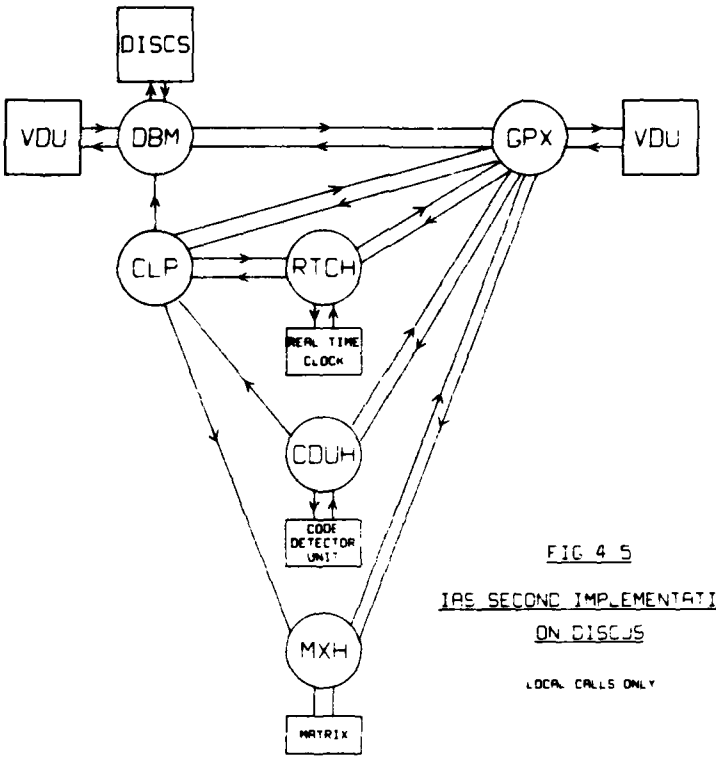


FIG 4 5

IAS SECOND IMPLEMENTATION ON DISCUS

LOCAL CALLS ONLY

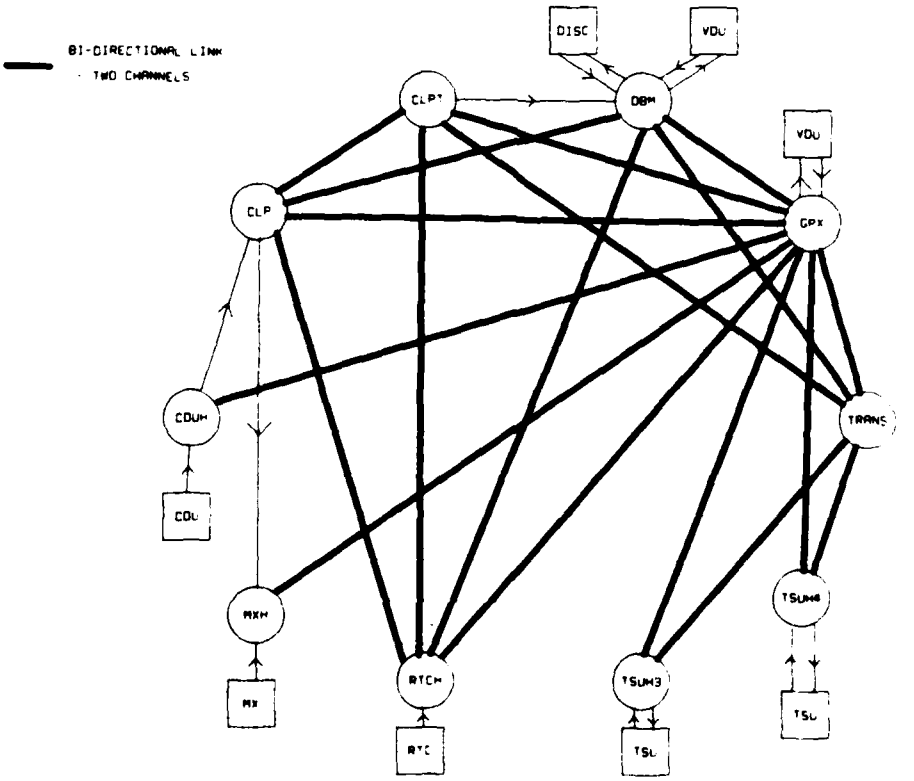


FIG 4 6 IAS FUNCTIONAL CONNECTIVITY - INCLUDING TRUNK CALL PROCESSING

5 SOFTWARE DEVELOPMENT AND TESTING

Having decided what processes are required to do the job of the IAS, and how the code on which these processes run is to be mapped on to processors, the job of designing and writing the code and constructing the data objects can begin. The design of the IAS software was based on previous experience and this report does not detail this design. Nevertheless, some useful points concerning methods of designing and writing applications software, (especially for a multi-processor machine) have emerged from the exercise, and the purpose of this section is to describe these.

5.1 Design of Applications Software

Having considered all the matters discussed in section 4, we now know what DISCUS functions are required, and what DISCUS channels are needed to connect them together. This is summarised in fig 4.6. Taking section 4 in conjunction with the control sequences described below, we also have some idea of the global data objects which will be required, though this will become clearer as the software design progresses. The next stage in the design procedure is to write down, for the processes supported by each DISCUS function:

(a) What tasks the process must perform
(b) What other processes it communicates with
(c) What messages it sends and receives
(d) What global arrays it accesses, and how (read only, read/write)

We can now invent a set of IAS inter-process message formats, and stipulate that processes will send only these messages.

Once this has been done, we can turn our attention to control sequences representing the operations which the switch has to perform. These were shown as a set of high level flow diagrams describing the operation of the switch, and were divided into sections, each section representing a DISCUS function in IAS. The codes of practice used in the preparation of the control sequences were as follows:

(a) Since the code for each DISCUS function will, in general, be segmented to facilitate writing and compiling, the control sequences should reflect this segmentation. Therefore, each page of control sequence diagrams (roughly as much as will fit on to one or two line printer pages on an easily readable scale), will represent a segment of code. Segments will be made small enough to allow this to be done.

(b) The structure of each page of control sequence diagrams will be such that, when manually following through any of the operations described on a page, the sequences on that page are entered at only one point. This rule of one entry point per page, (and correspondingly, one entry point per segment), both simplifies manual thread following on the control sequence diagrams, and improves the comprehensibility of the code when it is written. Code segments and control sequence pages should be labelled so that it is obvious which page corresponds to each segment.

(c) The level of detail in the control sequence diagrams will be such that the corresponding code can be written from them without very much further design effort, leaving the programmer free to concentrate on the coding details. This is a fairly vague definition of the level of detail required, but it was found that, after a little practice, control sequences could be prepared from which code could be written almost mechanically.

5.2 Writing IAS Applications Software

In order to be able to use the DISCUS system generator[1] to produce a loadable DISCUS function, the following information must be prepared:

(a) The code for the main segment of the function, and for all its subsegments, written in Coral for the Intel 8080 microprocessor, with the DISCUS language extensions.

(b) A global declarations file, which formally declares all the objects in the global data-base.

(c) A global equivalence file, which relates the local names for the global objects, specified within the function, to the names in the global declarations file.

The codes of practice used in the actual writing of the applications software were merely the standard structured programming practices normally employed, and need not be listed here. The only item worth noting is that data in global store was always accessed via the DISCUS operating system procedures provided for the purpose. It is, in fact, possible in the Mk I DISCUS system to access global store without using the DISCUS access procedures, if one is prepared to be sufficiently devious. This is, of course, a highly dangerous thing to do, since all the consistency checks provided by the operating system are circumvented, and the probability of irrevocably corrupting the data-base is thereby greatly increased. Fortunately, it has only once (for test purposes) been necessary to do this, and even then the problem was solved by merely using existing operating system procedures normally only available within the operating system itself, so that the software protection mechanisms still operated. In later versions of DISCUS, attempts to access global store directly from the applications program are trapped by a hardware mechanism.

5.3 Building and Testing a System

Once all the necessary files have been prepared, a system is put together by means of the DISCUS system generator[1]. No more will be said about the system generator here, but it should be noted that usually the entire applications system will not be generated at once. Initially, probably only one DISCUS function will be available, and it is convenient if this can be built into a small system and tested by itself. Two things are necessary for this:

(a) Some means of providing the processes running on the function to be tested with the stimuli which they would normally receive from other processes, and monitoring the messages it sends in response.

(b) A means of rearranging the inter-function connectivity so that these artificial stimuli can be injected, and the responses monitored.

Requirement (b) can be achieved by modifying the global equivalence file for the function under test so that the output messages of its processes are sent to some existing monitoring functions. The stimulus injection and response monitoring in (a) are provided by the manual control and testing facilities in the GPX function. This is normally connected so that messages can be sent from it to any other function, and therefore the only connectivity rearrangement required is to connect the output channel of the function under test to the input of GPX. All this, of course, assumes the prior existence of GPX, which must therefore be the first function to be generated. It can be tested by looping its output channel back to its input channel by a suitable change to

its global equivalent file. GPX can also be used to inspect and modify the global data-base from the system console, and can therefore monitor the changes made to the data-base by the function under test. It is also useful in setting up the data-base in any desired state to check the response of the function to stimuli under those conditions.

As each function is tested, either individually or with other previously tested functions, the system is gradually connected in its working configuration by changes to the global equivalence files. Even when all the functions are present, the GPX function remains part of the system, so that errors can be reported to it at run time, and it can be used to conduct the further testing which will be required as more faults become apparent.

Two facilities provided by the DISCUS system were particularly useful during testing:

(a) The "normal reset" - By pressing a processor's normal reset button, the execution of the applications code it supports can be made to restart from its normal start address. This is a useful way of allowing the system to continue after one processor has crashed, and the cause of the crash investigated. It does, of course, only affect one processor.

(b) The "RECOVERY POINT" facility - This is described in the operating system report[1], but briefly it allows the applications programmer to select the point in a DISCUS function to which control will be passed after the operating system detects a global access error, (such as an attempt to write to an array open for read only etc.), or after the applications code has called the operating system procedure ABORT. The effect of either of these two events is to release all global objects which are booked to the Aborting processor in their current state, and to place an end of message marker in any sink booked to the processor before releasing it too. A jump then occurs to the location of the last executed call of the operating system procedure RECOVERY POINT, or by default, the function's normal start address. The facility is intended for use in recovery from faults at run time, but can also be used during testing to force a jump to a routine to print out an error message, or an endless loop which will prevent the processor concerned from modifying the state of the system, especially the data-base, while the fault is investigated.

    In normal use the procedure RECOVERY POINT would be called at the function's wait point, so that the effect of abort is to abandon the transaction being processed at the time the fault occurs, and to start again with the next transaction as if nothing untoward had occurred. This is fine for preventing faults from causing a system crash, (which is what it was designed for) but is inconvenient when testing a function because, unless some very comprehensive error logging system is used (which does not yet exist), the state of the system at the time of the fault will generally be lost. Functions written with this kind of recovery mechanism may have to have other calls of RECOVERY POINT inserted into them for test purposes, which means that the function tested and the function eventually used will be slightly different. This may or may not be significant, but it is a little untidy.

One DISCUS facility which has not been used much is the Test Harness. This is effectively a scheduling program which treats each DISCUS function as if it supported only one process, and schedules these processes to run on the MDS. This was used early on in the project, but was found to be unnecessary once hardware became available.

6 TRAFFIC GENERATOR EXPERIMENTS

6.1 Introduction

Having devised the IAS applications software in accordance with the principles stated in previous sections, it was necessary to assess the performance of the IAS, and so test the validity of our ideas on how applications software should be designed for multi-processor systems. In order to do this, some means of providing work for the IAS to do was required, so that its performance could be measured under various traffic loading conditions. A traffic generator which was used for this purpose is described in section 6.2. In particular we wished to observe the degradation of system performance with increasing traffic load, the occurrence of bottlenecks in the system when functions reached their maximum throughput, and the effect on system performance of attempting to eliminate these bottlenecks by duplicating the functions responsible for them.

System performance can be illustrated by plotting a graph of the number of calls per channel per minute actually processed by the IAS against an input parameter of the traffic generator known as the "mean set call attempt rate", or, more simply, the "set call rate". This is the rate, in calls per channel per minute, at which the traffic generator would make calls if the IAS could process them instantaneously. In an arrangement such as the IAS/traffic generator system, no information can be lost as a result of the finite call processing rate, since a call cannot be initiated on a channel until the previous call on that channel has been completed. What happens is that the traffic generator is prevented from making calls at the set rate by the fact that the IAS cannot process them fast enough, and therefore makes calls at some lower mean rate. We would therefore expect the mean number of calls processed per minute to be roughly equal to the set call rate at low call rates, and the ratio (achieved call rate/set call rate) to decrease as the set call rate increases. Eventually the achieved call rate will become constant and independent of the set call rate, and the system is then said to be saturated. The value of the achieved call under these conditions is called the saturation level of the system. These effects have been observed, and are illustrated in figures 6.1 and 6.2. Evidently the more processing power IAS has, the greater the rate at which it can process calls, so that duplicating the busiest DISCUS function in the system will increase the saturation level of the switch. This effect is also shown in figures 6.1 and 6.2.

The degree of improvement to be expected from each functional duplication can be predicted theoretically as is shown in Appendix B. To make this prediction we need to know the mean time each function takes to process a transaction, and the number of processors supporting each function. In the version of IAS in which the CDU handler, matrix handler and RTC handler are all in one function, the mean time taken to process a device handler transaction depends on the rate at which calls are being attempted. This is because three different types of transaction are involved, which arrive at two separate inputs to the device handler function. These different types of transaction take different times to process, and the number of transactions of one type which can occur in each call is different from the number of transactions of other types. Furthermore, transactions arriving at one input (the DISCUS source) are processed preferentially. All of this makes the measurement of the mean time taken to process a transaction very difficult, so that these measurements were only performed for the version of IAS using three separate device handler functions, as shown in figure 4.5.

From these measurements, and knowing the number of processors supporting each function, we can predict the throughput of the system (in transactions/minute) using the equations derived in Appendix B and thus predict the saturation curves for IAS using different numbers of CLP (Local Call Processing) processors. These theoretical curves are shown, together with the corresponding experimental results, in figures 6.3 a to e.

6.2 Description of Apparatus

The apparatus consisted of the IAS, configured as in figure 4.4, and connected via a bi-directional 512 kb/s multiplexed link (of the type described in section 3.1), to a traffic generator. This consisted of a code generator and code detector unit similar to those used in IAS, controlled by a Plessey S250 computer. The software running on the 250 caused signals which simulated a number of local subscribers all making telephone calls to each other, at random intervals and at a set mean rate, to be sent to the IAS .

The traffic generator parameters were as follows:

(a) Mean call attempt rate. The traffic generator attempts to make calls randomly, but at a mean rate which can be set by the experimenter in increments of 1 call per channel per minute over the range from 1 upwards. This rate is used by the traffic generator to compute the interval which would be required between the start of a call originated by a subscriber on his channel and the start of the next call originated by the same subscriber, if calls were to be made at half the set call rate. This value is then multiplied by a random number between 0 and 1 (obtained by the use of a long PRBS), and the result used as the interval between successive call originations on the same channel. The distribution of the random number is approximately rectangular, its mean value being 0.5.

(b) Number of active channels. An active channel is defined as one on which calls can be originated, as distinct from one which may only terminate calls. In the experiments described here, the traffic generator was set to mimic 5 subscribers, all of which could both originate and terminate calls, that is, all 5 channels were active. The reason for the relatively small number of channels was that the IAS data-base contained only enough context information to support 5 local channel based processes.

(c) Time between start of ringing and called subscriber answer. This was set to 1.5 s.

(d) Time call spends in traffic. This was set to be 1.0 s.

Neither of the above two parameters was expected to be of great significance in the measurement of IAS performance.

(e) Times traffic generator waits for expected responses. If the traffic generator does not receive an expected response within a given time, it assumes that a fault has occurred. The lack of response is recorded, and the traffic generator attempts to clear down the call. The timeout periods were set to be sufficiently long that the response made by the IAS was always received by the traffic generator even under the heaviest loading conditions, except when IAS had in fact failed.

(f) Duration of traffic generator run. This could be set in increments of one minute in the range one minute upwards.

The traffic generator recorded all the above information, together with:

(a) The time taken to complete each call, measured from the time the traffic generator sends SEIZE to the time it receives the final RELEASE.

(b) How each call ended - OK, BUSY, PRE-EMPTED, etc.

(c) The total number of calls attempted during the run.

(d) The maximum, minimum and mean time taken to obtain a reply from IAS for each type of code sent.

(e) Any wrong responses received from IAS, or lack of response.

(f) Any errors in the traffic generator communications hardware.

6.3 Method of Experimenting

6.3.1 Measurement of Mean Achieved Call Rate

With IAS configured as in figure 4.4 (that is, with a single device handler function) the traffic generator was set to attempt calls at a mean rate of 1 call per channel per minute, and run for sufficiently long that about 2000 calls were made (about 6.5 hours). The run duration and the number of calls made were recorded. The experiment was repeated for set call rates up to that required to cause saturation. The runs were each long enough for about 2000 calls to be made, this number of calls being required to reduce the statistical variation in the mean achieved call rate to an acceptable level, see Appendix C. The whole set of runs was repeated using 2,3,4 and 5 processors supporting the CLP function.

The experiment was then repeated with IAS configured as in figure 4.5.

6.3.2 Measurement of Transaction Processing Time

As explained in section 6.1, the mean time taken to process a transaction is not readily measurable for IAS configured with one processor handling several devices. The measurements described in this section were therefore performed only on IAS configured with the CDU, matrix and RTC handlers on separate processors, as shown in figure 4.5.

The functions whose transaction processing times we are interested in measuring are those which lie on the "critical path" through the system. This path is the series of functions through which a transaction must pass in going from the input of the system to the output. If there are several paths, the situation is much more complicated and beyond the scope of this discussion. In the case of IAS the critical path consists of the CDU handler, local call processing, and matrix handler functions, as shown in figure 6.5. This assumes that we are only considering local calls, which is the case for the current version of the traffic generator.

For each of the functions of interest, the time taken to process a transaction can be defined as the time elapsing between the processor supporting the function finding a transaction at its input, and the next time it scans its input to look for a transaction. These times were measured using a logic analyser connected to the local bus of the processor concerned. Since, in the

IAS, transactions processed by any function do not all necessarily take the same time, a large number of measurements were taken, and a mean calculated. The number of measurements needed to produce a consistent mean value in repeated sets depended, of course, on the variability of processing times in the functions. for CDUH and MXH about 30 measurements were sufficient, while CLP needed about 600.

6.4 Results

6.4.1 Achieved Call Rate

If M calls are completed in a traffic generator run lasting T_R minutes, and there are n active communications channels, then the mean achieved call rate, R_m, is defined by:

$$R_m = \frac{M}{nT_R} \quad \text{calls/chan/min}$$

Five sets of traffic generator runs have been performed using 1,2,3,4 and 5 processors supporting the CLP function, each set consisting of runs at set call rates from 1 call per channel per minute to saturation level. A value of R_m was calculated for each run, and these values are plotted against set call rate, R_s, in figures 6.1 and 6.2. These refer to the IAS configured with a single device handler function and several device handler functions, respectively.

As can be seen, the use of two CLP processors almost doubles the maximum achievable call rate, and the use of 3, 4 and 5 CLP processors produce successively smaller increases. This is in agreement with the theoretical curves also shown in figures 6.3, and is explained in section 6.4.3.

6.4.2 Mean Transaction Processing Time, t

As explained above, these were obtained by the use of a logic analyser attached to the local bus of the processor supporting the function. The mean times obtained from these measurements were:

function	t (ms)
CDUH	10.0
CLP	171.3
MXH	57.0

6.4.3 Maximum Transaction Processing Rate

Under conditions where the instantaneous transaction arrival rate at the IAS input is equal to the maximum attainable transaction processing rate, the system is said to be saturated. This occurs at high set call rates, as can be seen from figures 6.1 and 6.2. The number of calls per channel per minute which IAS is actually processing under these conditions is constant and equal to the mean achieved call rate at saturation, R_{msat}, which can be read from the graphs. We know that, at saturation, all the simulated subscribers in the traffic generator will be originating calls (i.e. sending transactions to IAS) continuously, with no delay between one transaction and the next on the same channel, except that due to the time taken for IAS to respond. This leads to the conclusions that:

(a) At saturation, the length of the queue of transactions at the IAS input is equal to the number of active channels (simulated subscribers). Therefore, if there are n channels, the time elapsing between the arrival of a transaction at the IAS input and the sending of a response to it will be equal to the time taken to service n transactions. (This includes time spent in both processing and waiting to be processed.)

(b) Since all simulated subscribers are originating calls as fast as the response time of IAS will allow, all the calls will end BUSY.

If there are x_B (in IAS this is equal to 14) transactions in a call ending BUSY, then the maximum transaction servicing rate, or throughput, of IAS, m_t, can be predicted from:

$$m_t = n\ x_B\ R_{msat} \qquad \text{trans/min}$$

which is derived in Appendix B.

With IAS configured as in figure 4.4, this gives the following results:

Number of CLP processors, k	m_t (trans/min)
1	322
2	539
3	609
4	620

and with IAS configured as in figure 4.5:

Number CLP processors, k	m_t (trans/min)		k/t_{CLP}
	measured	predicted	
1	373	350	350
2	620	700	700
3	823	924	1050
4	922	970	1400
5	956	975	1750

where t_{CLP} is the time taken for CLP to process a transaction, in minutes. Measured values of m_t are correct to within $\pm$ 0.05 trans/min in the worst case.

The results in the above table are shown graphically in figure 6.4. k/t_{CLP} is the rate at which CLP would process transactions if the queue at its input always contained at least one transaction, so that no CLP processor was ever idle. Note that m_t is equal to k/t_{CLP} when only one or two CLP processors are used, but that m_t becomes smaller than k/t_{CLP} as more CLP processors are used. This can be explained as follows. From our measurements of t (for IAS configured as in figure 4.5), we know that CLP supported by a single processor can process transactions only at about one third of the rate of the second slowest function, MXH. If only one CLP processor is used, then under saturation conditions CLP will be by far the slowest function, and will have a queue at its input which always contains at least one transaction, so that it will never be idle. If we increase the number of CLP processors to two, this argument is still true. However, if we use three or more CLP processors, CLP can then process transactions at a rate which is of the same order as those of the other functions on the critical path, and so the queue at its input is more often zero, so that a proportion of CLP's time is spent idling. This, of course, limits the system throughput to less than what could be achieved if

CLP (and all the other functions) were always busy, and explains the effect of using multiple CLP processors on the IAS saturation curve. Since CLP is almost never idle if one or two processors are used to support it, we would expect the mean achieved call rate at saturation for IAS with two CLP processors to be roughly double its value for one CLP processor, which is indeed what happens. However, as we use more than two CLP processors, a progressively greater proportion of their time will be spent idling, so that progressively smaller increases in the achieved call rate at saturation can be expected. This is again confirmed by experiment.

However, for the system using several device handlers running on the same processor we find that the use of four CLP processors gives virtually no improvement compared to the use of three. This is probably because the device handler function is almost never idle when three or more CLP processors are used, and thus constitutes the same sort of bottleneck as CLP does if only one processor is used for it. The difference is that this bottleneck cannot be alleviated by duplicating the device handler processor, since the DISCUS hardware architecture does not allow the duplication of peripheral handlers. This emphasises the point made earlier about keeping device handlers small and fast, so that this problem does not arise. Splitting up the single DH function into several separate DISCUS functions, each handling only one device, both increased system throughput and allowed much more efficient use to be made of four CLP processors.

Note that no improvement in throughput beyond what can be achieved with five CLP processors is possible by adding more copies of CLP. If five CLP processors are used, CLP can then process five transactions simultaneously, and since there can never be more than five transactions in the system at the same time if there are only five active channels, the use of six or more CLP processors will result in at least one CLP processor being idle at all times.

6.5 Conclusions

It has been found that the throughput of a DISCUS based system can be increased by function duplication. By measuring the mean time for each function to process a transaction it is possible to determine which function should be duplicated to give the greatest increase in throughput, and to predict quantitatively how much improvement will be achieved by the use of given numbers of processors supporting each function on the system's critical path. Knowing the maximum number, n, of transactions which can be simultaneously present in the system we can predict that no further increase in throughput can be achieved by duplication of functions on the critical path than that obtained by the use of n processors supporting each such function.

The mathematical model of the operation of compelled signalling systems developed in Appendix B has been tested by comparing the predicted saturation curves with those produced from experimental results. The model is fairly accurate for values of R_S well below that required to cause saturation, but predicts values of R_M which are too high by up to 12% near saturation. The exception to this is the curve for 1 CLP processor, in which the predicted R_M values are slightly too low. No satisfactory explanation for these discrepancies has yet been found. The model may be more generally applicable, and this could be the subject of further research.

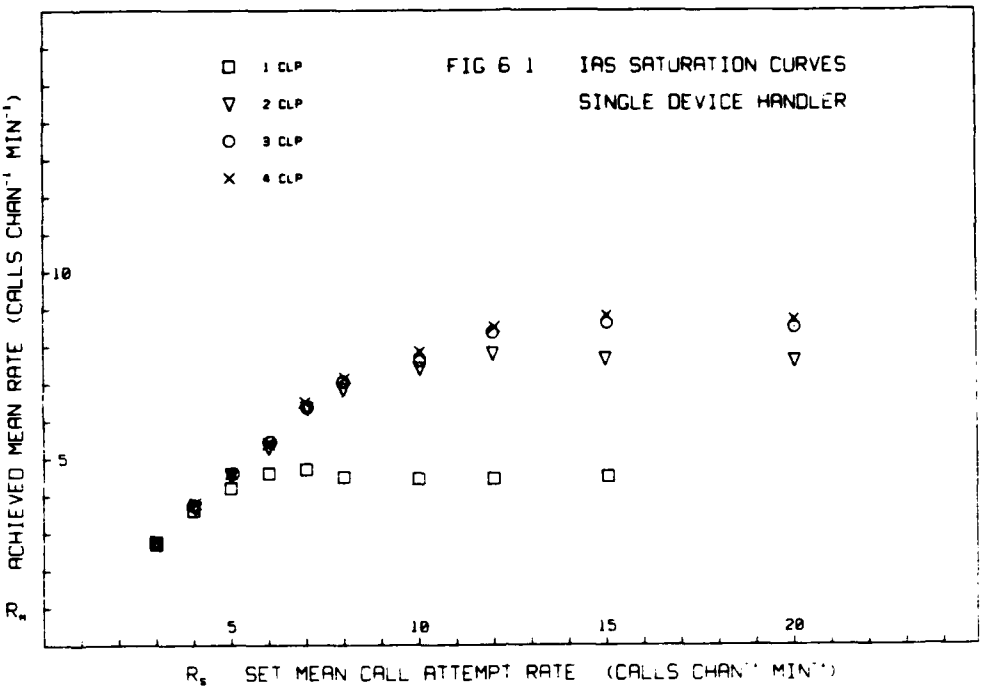


FIG 6 1 IAS SATURATION CURVES
SINGLE DEVICE HANDLER

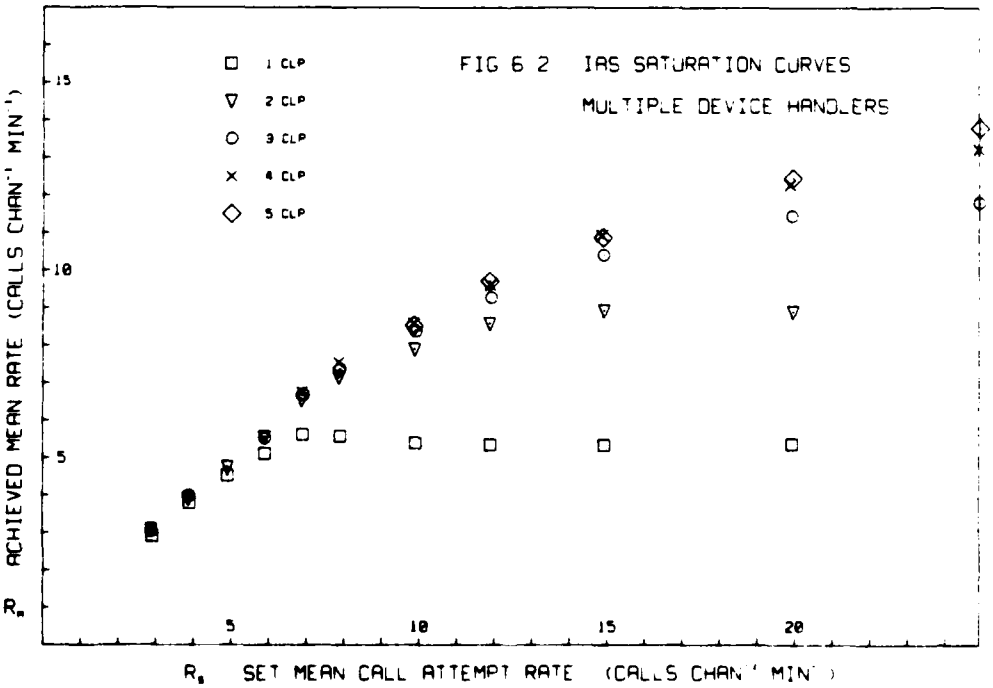


FIG 6 2 IAS SATURATION CURVES
MULTIPLE DEVICE HANDLERS

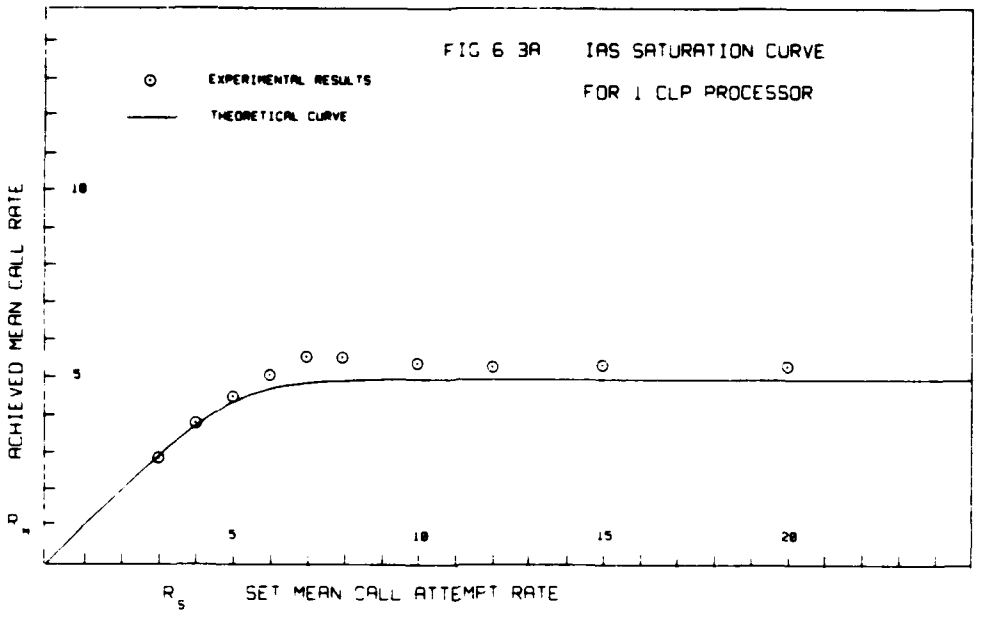


FIG 6 3A IAS SATURATION CURVE
FOR 1 CLP PROCESSOR
EXPERIMENTAL RESULTS
THEORETICAL CURVE
ACHIEVED MEAN CALL RATE
10
5
5
10
15
20
R_a
R_s SET MEAN CALL ATTEMPT RATE

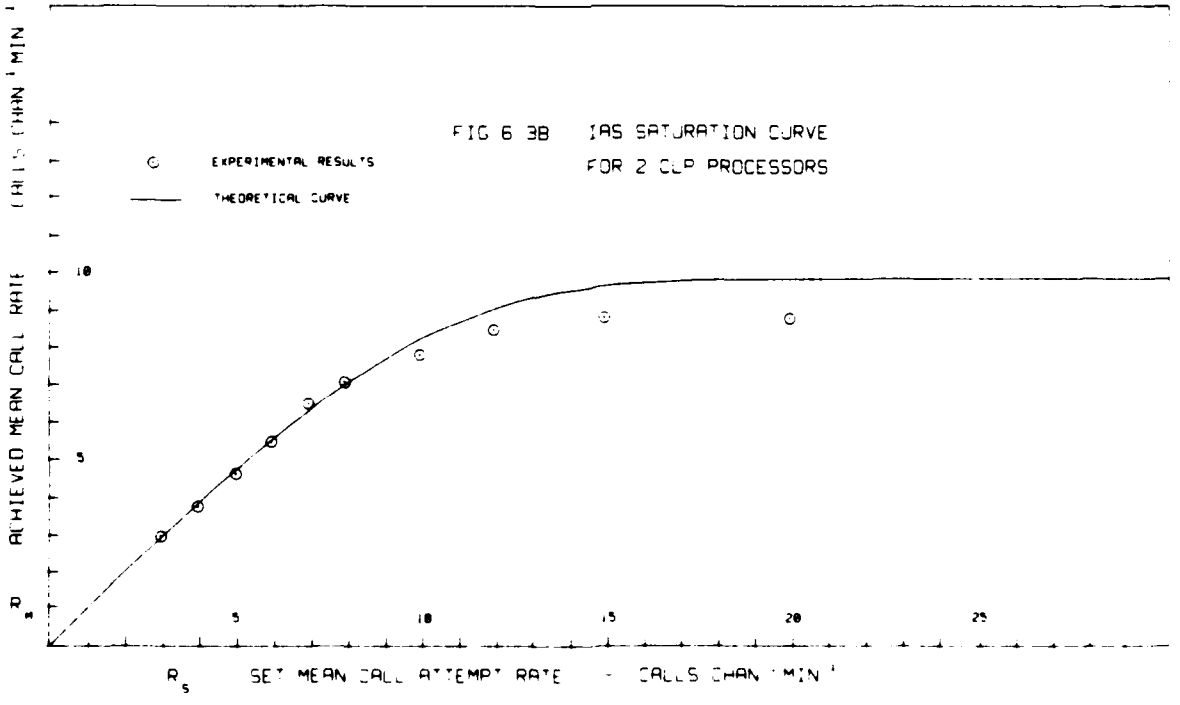


FIG 6 3B IAS SATURATION CURVE
FOR 2 CLP PROCESSORS
EXPERIMENTAL RESULTS
THEORETICAL CURVE
ACHIEVED MEAN CALL RATE
10
5
5
10
15
20
25
R_a
R_s SET MEAN CALL ATTEMPT RATE - CALLS CHAN^{-1} MIN^{-1}

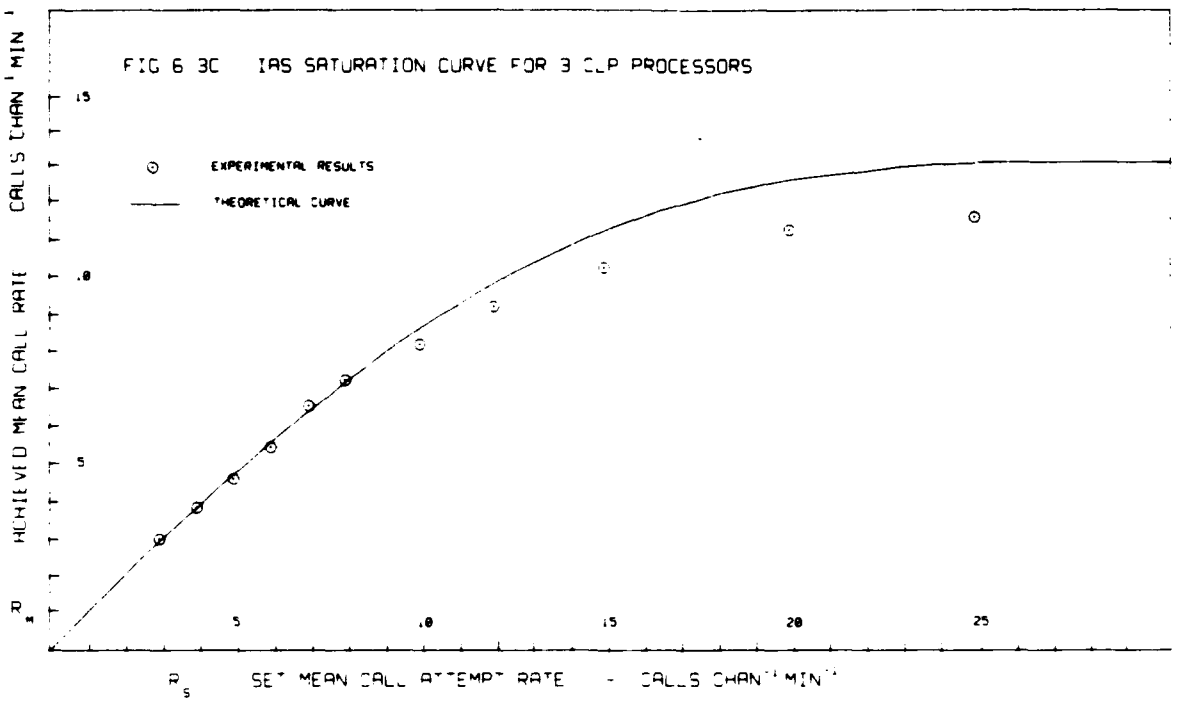


FIG 6 3C IAS SATURATION CURVE FOR 3 CLP PROCESSORS

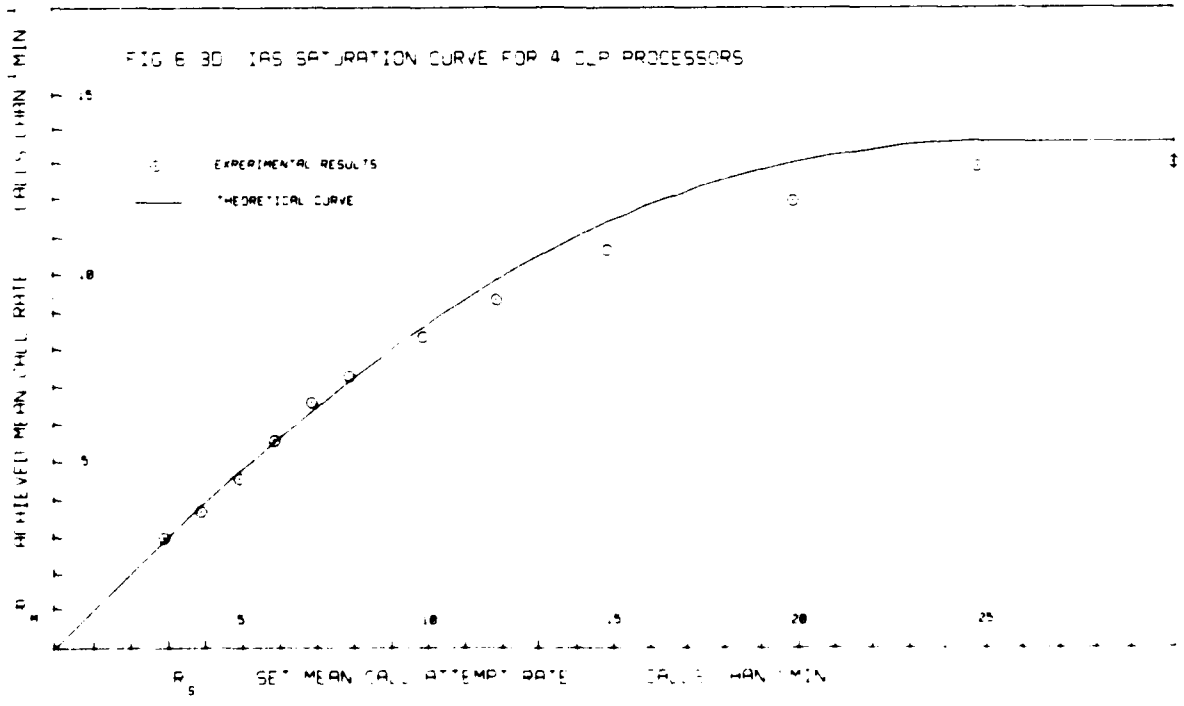


FIG 6 3D IAS SATURATION CURVE FOR 4 CLP PROCESSORS

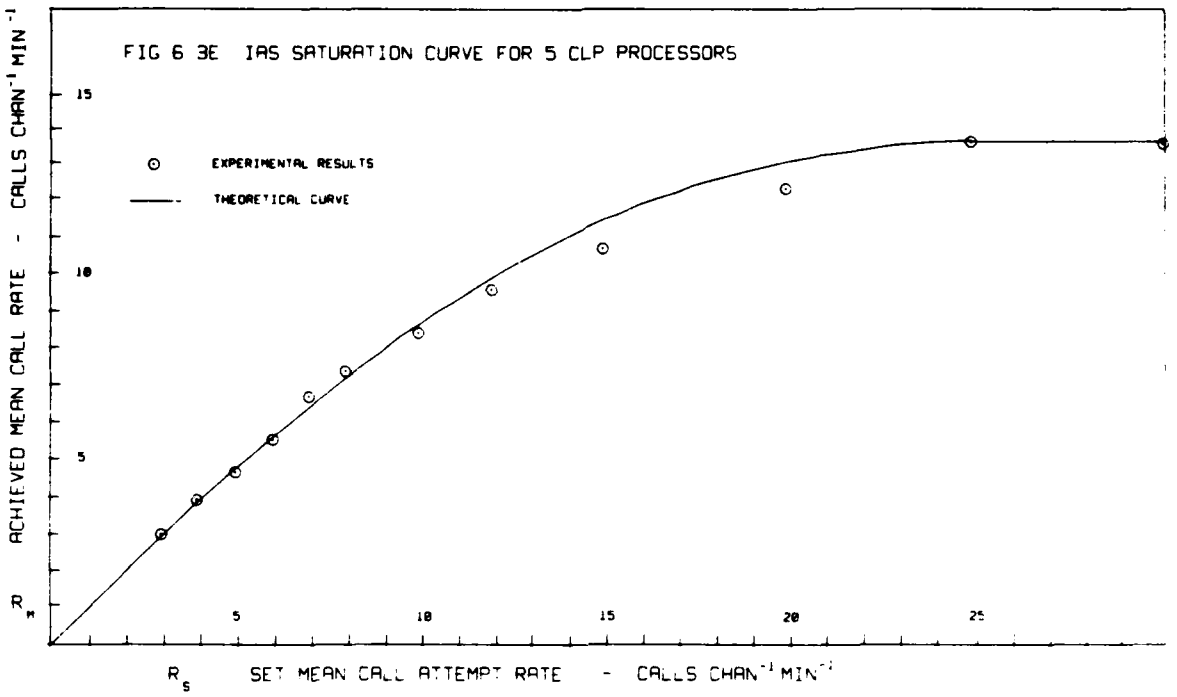


FIG 6 3E IAS SATURATION CURVE FOR 5 CLP PROCESSORS
EXPERIMENTAL RESULTS
THEORETICAL CURVE
ACHIEVED MEAN CALL RATE - CALLS CHAN⁻¹ MIN⁻¹
SET MEAN CALL ATTEMPT RATE - CALLS CHAN⁻¹ MIN⁻¹

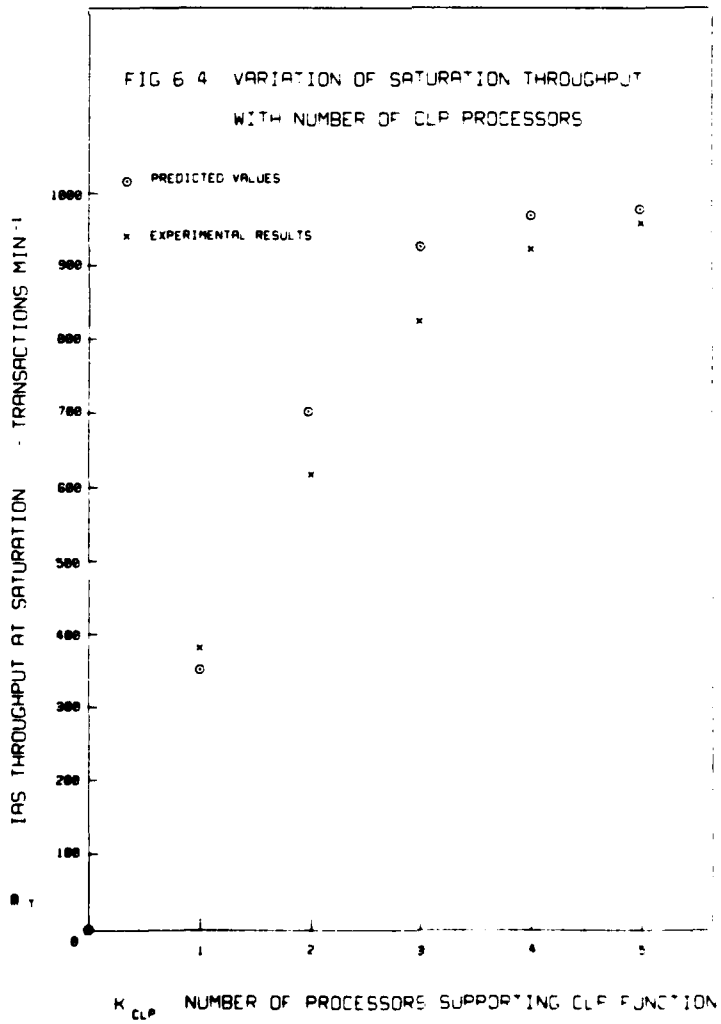


FIG 6 4 VARIATION OF SATURATION THROUGHPUT
WITH NUMBER OF CLP PROCESSORS
PREDICTED VALUES
EXPERIMENTAL RESULTS
IAS THROUGHPUT AT SATURATION - TRANSACTIONS MIN⁻¹
NUMBER OF PROCESSORS SUPPORTING CLP FUNCTION

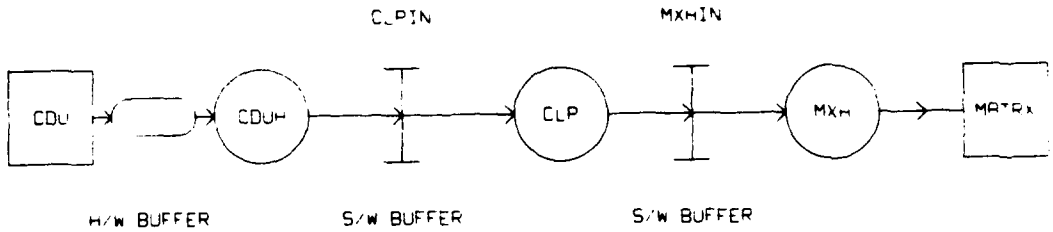


FIG 6 5 FUNCTIONS AND BUFFERS ON IAS CRITICAL PATH

7 CONCLUSIONS

In conclusion, let us return to the aims of the project as stated in the introduction, and see how far they have been achieved.

7.1 Viability of DISCUS in Large Applications

It has, evidently, been possible to use the DISCUS hardware and operating system as a convenient foundation for the IAS applications software. However, the way in which DISCUS was employed was, of necessity, somewhat different from that envisaged by its designers. There are several reasons for this.

(a) It was found necessary to abandon the concept of having only one process per processor, since the run-time structure devised for IAS required far more processes than there were processors on which to run them. As soon as more than one process can run on a processor, the protection of process context provided by the DISCUS hardware architecture is lost, since all processes running on the same processor can physically access all of that processor's local store. In IAS, many of the processes are of the same type, and can therefore be supported by the same code, so that attempting to run only one type of process on each processor is an attractive option, (though even this was not always possible given the small number of processors available).

(b) Having sacrificed the hardware context protection, we can, without any further penalty, improve the throughput of the system by duplicating any DISCUS function which is causing a bottleneck, so that, if the duplicated function supported only one process type, we now have more than one processor per process type. What we cannot allow is the running of the same actual process on two or more processors simultaneously. This would, at best, be a waste of computer time, if all the processors performed exactly the same operations, and could have entirely calamitous consequences if the operations were different. The IAS applications software is constructed to avoid this unhappy state of affairs. All of this leads to the emergence of a conflict between two of the advantages claimed for DISCUS. It is possible either to use the DISCUS architecture to protect process context, or to increase throughput by functional duplication, but it is not possible to do both of these for the same DISCUS function.

(c) Another advantage of DISCUS, used as originally intended, is that there is no requirement for a scheduler. Of course, once we have more than one process per processor, some form of scheduler is needed, and if the processes are of more than one type, this can become quite complicated. Moreover, since the DISCUS operating system does not include a scheduler, this must be written as part of the applications software, which implies that scheduling in DISCUS functions written by different programmers will be different. This is a little inelegant.

7.2 Advantages and Disadvantages of the DISCUS Environment

As far as software design is concerned, the DISCUS multi-processor environment is very similar in many respects to that provided by a multi-processing operating system on a single processor machine. DISCUS does, however, provide two advantages over single processor systems:

(a) It is possible to do some jobs simultaneously on different processors, thereby increasing throughput compared to what can be achieved using only a single processor. Care must, however, be taken in choosing these jobs, so as to as to avoid undue software complexity arising from the need to sort input data and collect results in matching sets.

(b) A particularly convenient application of this parallel processing ability is functional duplication. We have discovered how to determine which DISCUS function to duplicate to achieve the greatest increase in throughput, and how much improvement is to be expected under various conditions. However, in applications where no more than a certain number of transactions of a given type can exist simultaneously, there is no point in providing more than this number of processors running the same process type. This sets a limit on what can be achieved by duplicating functions in this type of application.

The remaining comments apply equally to single and multi-processor systems.

(c) The DISCUS operating system formalises inter-process communication, thereby reducing the chance of errors occurring.

(d) The DISCUS System Generator formalises system building, and facilitates the testing of some DISCUS functions in the absence of others by allowing connections between functions to be re-routed easily by manipulation of the global equivalence files. Used in conjunction with the IAS Manual Control and Testing Facility (GPX) this makes testing and debugging fairly simple.

(e) Testing as in (d) above assumes the prior existence of GPX. If this is part of the applications software, it must therefore be the first DISCUS function to be developed, and can be tested by looping its output channels back to its input by a suitable simple modification to its global equivalence file. However, the GPX software used in IAS was written sufficiently generally that it could be used with other applications, provided that the names and other characteristics of the global objects were changed as necessary. If the data-base editing and message handling routines were part of the DISCUS foundation software, a GPX suitable for use with any given application could be produced by the System Generator, thus reducing the amount of applications software to be written and tested, so that systems could be developed more quickly.

7.3 Guidelines for Applications Software Development

It has been possible to formulate a set of guidelines to assist in:-

(a) Choosing a process structure for an application.
(b) Performing a functional decomposition of an application.
(c) Mapping the resulting software on to a multi-processor system.

These have been set out in section 4.4, and there is no need to repeat them here. Two additional points are worth re-stating, however.

(d) Since both deciding on a process structure and performing a functional decomposition are, it seems, largely independent of the way in which the resulting applications software is mapped on to the available processor(s), the use of several processors and/or a DISCUS-like operating system does not enforce the division of the software into manageable parts. This remains a discipline which the applications designers must apply themselves.

(e) If control sequences are designed carefully, software can be written from them almost mechanically. This is probably because most of the decisions concerning what must be done to what data in what order are taken at the control sequence design stage, leaving the programmer free to concentrate on the use of the language when the time comes to actually write the code. It must, however, be admitted that there may be applications for which it is much more difficult to design suitable control sequences than it was for IAS. Given that they can be designed, the use of a computer to store the sequences, and a screen-editor to display and modify them in flow chart format has been found useful, though the initial data entry is somewhat tedious. The use of computer graphics techniques, possibly including editing by light-pen, may well offer a considerable improvement by speeding up the initial typing in, and allowing a larger section of the sequences to be displayed than is possible on a VDU screen.

7.4 Final Comments

The usefulness of the DISCUS hardware and operating system as a foundation for a fairly large and complex software development project have been amply demonstrated, although the structure of the application was, perforce, somewhat different to that envisaged by the original designers of the operating system, in that the "one process per processor" approach was found impracticable, and had to be abandoned.

The list of guidelines for software development which have been drawn up as a result of this work are, in part, yet another example of "Codes of practice for Structured Software", and as such merely present a familiar view from a slightly unusual viewpoint. The viewpoint will, however, probably be similar for many multi-processor systems, so that the guidelines should apply to environments other than DISCUS.

The mathematical model of the IAS/Traffic Generator is not really concerned with the properties of multi-processor systems, but it has led to an appreciation of the way in which the maximum benefit can be obtained from functional duplication. More importantly, it has allowed a certain insight to be gained into the behaviour of communications systems using compelled signalling, and may have applications beyond IAS, but this is more properly the subject of further work and a further report.

8 REFERENCES

[1] M P Griffiths,
"The DISCUS Operating System",
RSRE Report 82009

[2] H S Field-Richards,
"The DISCUS Hardware System",
RSRE Report 82010

[3] T G Connelly,
"IAS Communications Hardware and Peripheral Interfaces",
(in preparation)

[4] A L Simcock,
"A Microprocessor Controlled Code Detector Unit",
T2 internal paper 1980 (unpublished)

[5] G R Holder,
"A Microprocessor Based Trunk Signalling Unit",
T2 internal paper 1979 (unpublished)

[6] K Jackson and H R Simpson,
"MASCOT",
RRE Tech. Note 778

[7] H S Field-Richards,
"The DISCUS Hardware Descriptions and Appendices",
RSRE Memo 3483

[8] E Kreyszig,
"Advanced Engineering Mathematics",
Wiley, 1968

9 ACKNOWLEDGEMENTS

In the course of the IAS project, several people contributed ideas on the design of the software, and many more helped in developing the hardware. In particular we would like to thank the following for their assistance:

Ian Outram, who was for a considerable period in charge of the project, and with whom we had many useful discussions on the design of software for systems like IAS.

Mike Partridge, who in the course of a number of lengthy discussions, prompted us to think more precisely about the meaning of the results of the Traffic Generator experiments, and about the validity of the mathematical model. In particular, Mike made us realise that, as usual, things were not as simple as they appeared.

Fred Bell, who wrote the original version of the Traffic Generator.

Peter Bottomley and **Richard Hawley**, for their efforts in modifying the Traffic Generator for use with the IAS, and for carrying out frequent alterations and additions to it to meet our changing requirements.

Steve Bracking, who developed the LDA, which was an essential tool in the testing and debugging of the IAS local call software.

Hugh Field-Richards, who developed the DISCUS multi-processor hardware which supported the IAS.

Howard Nichols, who originally devised the DISCUS operating system.

APPENDIX A

IAS Mark I - The Great Mistake

This appendix describes briefly the structure of the first attempt to map the IAS software on to the DISCUS machine. This was before most of the guidelines presented in section 4.4 had been formulated, and shows how neglect of some areas of software design in favour of others can lead to a hopelessly unworkable scheme.

The rationale for the structure went something as follows. We had a job which had to be done (making telephone calls) and a machine with the experimental distributed architecture to run the job on. It therefore seemed most sensible to break the job up into its obvious component parts (collecting digits, handling the matrix, clearing down calls and so on) and allocate a notional "process" to each of them. Since we should keep inter-computer communication down to a minimum for reasons of efficiency, it was decided to put as much processing as possible in to the device handlers, and also amalgamate most of the processes which actually carried out some identifiable task (as opposed to analysis) in to another function. Finally, also for reasons of efficiency, there was to be no global data-base, all processors would keep their data in local store, and global store would only be used to pass messages. This gave the structure illustrated in figure A.1.

In the light of later knowledge, this structure is very suspect. Here is a list of its more obvious flaws:

(1) No data-base in global store means that none of the functions can be duplicated.

(2) The device handlers (in particular CDUH) have been given far too much to do. They are prime candidates for an unrelievable bottleneck.

(3) The control structure is particularly poor. Having no global data-base that processes can look at to check on the state of each call, means that each process has to keep a record itself of how far each call has progressed. This means that there is the possibility (probability?) that these records will become inconsistent, and it would be very difficult to reconcile them. Even, for example, in the comparatively common circumstance where "Release" is received half way through a call, the CDU handler must be aware of the state of the call in order to decide where to pass the message. There was some attempt made to have a centralised "state of call" by the use of the Call Management Control function, but the necessity to communicate with it only by messages makes the implementation extremely cumbersome.

Whilst this arrangement could probably have been made to work by the addition of a properly organised data-base, the process structure does not reflect the easiest way to implement a simple local call, which is essentially a serial task. Moreover, there was far too much emphasis on the breaking up of the local call management process, forgetting that the IAS has a lot more to do than just connecting local telephones (data-base management, trunk calls, response to requests from other switches, etc) so that finely dividing local call processing is neither necessary nor desirable.

In summary then, this could probably have been made to work, but would have been very difficult to get right and would have made life much more difficult when attempting to evaluate functional duplication.

APPENDIX B

Mathematical Model of IAS/Traffic Generator System

B.1 Introduction

In this appendix we attempt to predict the saturation curves for IAS, that is, the curves obtained by plotting the mean achieved call rate against set mean call attempt rate for different numbers of processors supporting the CLP function.

The original idea was to present a "black box" model, making no assumptions about the detailed queuing phenomena within IAS, but it proved impossible to complete the analysis on this basis. An examination of queuing in systems consisting of a number of functions in series has also, therefore, been included.

In predicting the saturation curve, we first analyse the interaction of IAS and the traffic generator, to derive a relationship between the set call rate, R_S, and the achieved call rate, R_M. It turns out that this relationship involves the mean time taken to complete a call, and some considerable labour is required to eliminate this parameter. We eventually derive a set of four simultaneous non-linear equations which require numerical solution to obtain the predicted saturation curves.

The final part of the appendix is concerned with predicting the existence of the saturation phenomenon, and with gaining confidence in the previously derived theoretical results by using them to predict a few properties of the system which we might expect from a simple common sense approach.

B.2 Traffic Generator Characteristics

The rate at which the traffic generator originates calls on any given active channel is determined by the set call rate provided as an input parameter, measured in calls per channel per minute. This set call rate is denoted R_S. The time between the start of one call originated by the traffic generator on a given channel, and the start of the next call originated on the same channel, (T_S), is computed by the traffic generator as:

$$T_S = \frac{2r}{R_S}$$

where r is a random number in the range 0 to 1 inclusive. The probability density function for r is rectangular, as is shown in figure B.1, so that the mean value of r is 0.5, and the mean value of T_S is hence given by:

$$\overline{T}_S = \frac{1}{R_S}$$

provided that the time taken to complete a call (T_C) is small compared to T_S. Note that T_C is the time between the start of a call and the end of the same call. The corresponding probability density function for T_S is shown in fig B.2.

By definition, the probability that T_S lies in an interval Δt wide within the range $0 <= T_S <= (2/R_S)$ is the area under curve in range t to $t + \Delta t$ and is given by:

$$P(t <= T_s <= t + \Delta t) = \frac{R_s}{2} \Delta t$$

Now suppose we have a situation in which T_c is not negligibly small. The probability density function for T_s is then as shown in fig B.3.

If r is randomly chosen such that $\frac{2r}{R_s} >= T_c$, then T_s will be $\frac{2r}{R_s}$

If r is randomly chosen such that $\frac{2r}{R_s} < T_c$, then T_s will be T_c

Probability that $\frac{2r}{R_s} < T_c$ = shaded portion of fig B.3 = $\frac{T_c R_s}{2}$

Therefore, probability that $T_s = T_c$ is $\frac{T_c R_s}{2}$

probability that $T_s > T_c$ is $\frac{(2/R_s - T_c)}{2/R_s} = (1 - R_s T_c/2)$

probability that $T_s < T_c$ is 0

The probability density function in fig B.3 is unusual in that, although the area under it must, by definition, be unity, the area under the curve in the range $T_c < T_s <= (2/R_s)$ is $(1 - T_c R_s/2)$, which is less than one. The area required to bring the total area under the curve to unity is "under" the spike at $T_c = T_s$, this area being equal to that of the shaded region of fig B.3.

The mean value of T_s is given by the general result:

$$\bar{T}_s = \int_{-\infty}^{\infty} T_s \, f(T_s) \, dT_s$$

We can represent $f(T_s)$ over its non-zero range ($T_c <= T_s <= 2/R_s$) in terms of a delta function:

$$f(T_s) = \frac{R_s}{2} + \text{area under spike} \times \delta(T_s - T_c) = \frac{R_s}{2} + \frac{R_s T_c}{2} \delta(T_s - T_c)$$

Hence:

$$\bar{T}_s = \int_{T_c}^{2/R_s} \left\{ \frac{R_s}{2} + \frac{R_s T_c}{2} \delta(T_s - T_c) \right\} T_s \, dT_s = \left[\frac{R_s T_s^2}{4} \right]_{T_c}^{2/R_s} + \frac{R_s T_c^2}{2}$$

or:

$$\bar{T}_s = \frac{1}{R_s} + \frac{R_s T_c^2}{4}$$

The mean achieved call rate, R_M, is given by:

$$R_M = \begin{cases} 1/\bar{T}_s & \ldots\ldots \quad T_c <= 2/R_s \\ 1/T_c & \ldots\ldots \quad T_c >= 2/R_s \end{cases}$$

or:

$$R_M = \begin{cases} \dfrac{4R_s}{4 + T_c^2 R_s^2} & \ldots\ldots \quad T_c <= 2/R_s \\ 1/T_c & \ldots\ldots \quad T_c >= 2/R_s \end{cases} \qquad \ldots\ldots (1)$$

B.3 IAS Characteristics

If we wish to predict the shape of the R_M versus R_S curve (the saturation curve), we need to find an expression for T_C in terms of R_M or R_S, or both. This is derived below. However, before continuing, let us define a few symbols.

x = Mean number of transactions which IAS must process for a call to be completed. A call is said to be complete when it has gone into traffic and later been cleared down, or, in the case of a busy call, been cleared down after dissuasion.

x_B = Number of transactions in a call which ends BUSY

x_0 = Number of transactions in a call which ends OK (cleared down after going into traffic)

We assume that all calls end either busy or OK; the minority which do not are ignored.

m = Throughput of IAS in transactions per minute

N = Mean number of calls simultaneously in existence

n = Number of channels on which calls can be originated. In the case of this experiment, this number was 5

Q = Mean number of transactions in the IAS at any time, including both those in the input queue and those being processed

The values of x_0 and x_B for IAS are:

$$x_0 = 22, \text{ and } x_B = 14$$

These are approximate as they include a correction for transactions which require no processing.

Consider the time taken to service a transaction arriving at the IAS input. There will, in general, be a queue of previously arrived transactions waiting to be processed. If the mean number of transactions in the system is Q, and the throughput of the system is m transactions per minute, then the mean time which a transaction spends in the system will be Q/m minutes. If a call consists of x transactions, then the mean time taken to complete a call, T_c, is given by:

$$T_c = \frac{Qx}{m}$$

This, of course, assumes that all the time a call takes to complete is due to the time taken to process transactions, or the time transactions spend waiting to be processed. In the case of the experiment, every call which ends OK includes 1.5 seconds during which the simulated called subscriber's subset is ringing, and 1 second spent in traffic. If the fraction of calls ending OK is F_{OK}, and we denote the times spent ringing and in traffic as T_R and T_T respectively, then we get:

$$T_c = \frac{Qx}{m} + (T_R + T_T)F_{OK} \quad \ldots . \quad (2)$$

We now need expressions for Q, x, m and F_{OK}.

B.3.1 Expression for Q

P(a given channel is originating a call) =

$$\frac{\text{mean number of channels simultaneously originating calls}}{\text{total number of channels}} = \frac{N}{n}$$

We can also write:

P(a given channel is originating a call) =

$$\frac{\text{mean time taken to complete a call}}{\text{mean time available for each call on this channel}}$$

$$= T_c/(1/R_M) = R_M T_c$$

So,

$$\frac{N}{n} = R_M T_c$$

or,

$$N = nR_M T_c$$

If the time taken for the traffic generator to process replies from IAS is small compared with the time taken by IAS to process a transaction, then all calls simultaneously in existence will have one (and because of the compelled signalling, only one) transaction in IAS at all times. That is, the number of transactions in IAS at any time is the same as the number of simultaneously existing calls, so Q = N. Hence,

$$Q = nR_M T_c \quad \ldots . \quad (3)$$

This is, in fact, Little's formula, well known from queueing theory.

B.3.2 Expression for F_{OK}

Probability that a channel is originating a call at any instant = N/n

Probability that a channel is terminating a call at any instant =

(probability that the channel is not originating a call) x
(probability that at least one of the N calls is to it)

Therefore,

P(a channel is terminating a call) =
$(1 - N/n)(1 - P(\text{none of the N calls is to it}))$

P(a given call is to a given channel) = 1/n

So,

P(a given call is not to a given channel) = $1 - 1/n$

P(none of the calls is to a given channel) = $(1 - 1/n)^N$

P(a channel is terminating a call) = $(1 - N/n)\{1 - (1 - 1/n)^N\}$

P(channel busy) = P(chan busy as originator) + P(chan busy as terminator)

Therefore,

P(channel busy) = $N/n + (1 - N/n)\{1 - (1 - 1/n)^N\}$

So,

P(chan not busy) = $1 - P(\text{chan busy}) = 1 - N/n - (1 - N/n)\{1 - (1 - 1/n)^N\}$

P(call ends OK) = P(destination channel not busy)
= fraction of calls ending OK
= F_{OK}

Hence,

$$F_{OK} = 1 - N/n - (1 - N/n)\{1 - (1 - 1/n)^N\}$$

$$= (1 - N/n)(1 - 1/n)^N$$

$$= (1 - R_M T_c)(1 - 1/n)^{nR_M T_c} \quad \ldots\ldots \quad (4)$$

since $N = nR_M T_c$

B.3.3 Expression for x

x = (fraction of calls ending OK)x_0 + (fraction of calls ending busy)x_B

$$= F_{OK}x_0 + (1 - F_{OK})x_B = F_{OK}(x_0 - x_B) + x_B$$

Using (4) to substitute for F_{OK} we get:

$$x = (1 - R_M T_c)(1 - 1/n)^{nR_M T_c}(x_0 - x_B) + x_B \quad \ldots\ldots \quad (5)$$

B.3.4 Expression for m

To obtain an expression for the system throughput, m, it is necessary to take a closer look at the internal structure of the IAS software. This consists of a number of functions, in this case three, connected in series. Let us call these functions f_1, f_2 and f_3. Each function f_i is supported by k_i processors, and the time taken for a processor supporting f_i to process a transaction is t_i minutes.

For several functions in series in a compelled system such as IAS, all functions will have the same throughput if we take an average over a sufficiently long period. If this were not so, queues at functions with lower throughputs would increase indefinitely, and this does not happen. However, the processing times, t_i, of the functions will, in general, be different. The throughputs are equalised by the fact that the mean queue length at each function adjusts itself so that the slower functions (those with high values of t) have proportionately longer queues, and so have a relatively low probability of having to idle while waiting for a transaction which they can process. Functions with lower t values have shorter queues, so that the probability of their having to idle is relatively high.

Consider a function, f, supported by k processors, each having a processing time t. We drop the subscripts to reduce clutter. The mean rate at which the function can service transactions is its throughput, and this is equal to m, the throughput of the whole system if it consists of functions in series. Let us denote the instantaneous servicing rate by Z, and the instantaneous length of the queue of transactions at this function, including both those waiting to be processed and those actually being processed, by q. We can then write:

$$Z = \begin{cases} \frac{k}{t} & \text{. . . . . } \quad n \geq q \geq k \\ \frac{k-1}{t} & \text{. . . . . } \quad q = k - 1 \\ \frac{k-2}{t} & \text{. . . . . } \quad q = k - 2 \\ \vdots & \\ \frac{1}{t} & \text{. . . . . } \quad q = 1 \\ 0 & \text{. . . . . } \quad q = 0 \end{cases}$$

where n = (maximum possible value of q) = (number of channels for IAS)

This represents a probability distribution for Z, the probability of each value of Z being the probability that q is in the corresponding range (or at the corresponding value). The mean of this distribution is m, the mean throughput, in other words m is equal to the mean of Z.

By definition, therefore:

$$m = \sum_{\text{all } Z} Z\,P(Z) = \sum_{q=0}^{n} Z\,P(q) \qquad \text{since } n \geq q \geq 0$$

Therefore:

$$m = \sum_{q=k}^{n} \frac{k}{t} P(q) + \sum_{q=0}^{k-1} \frac{q}{t} P(q)$$

$$= \sum_{q=k}^{n} \frac{k}{t} P(q) + \sum_{q=0}^{n} \frac{q}{t} P(q) - \sum_{q=k}^{n} \frac{q}{t} P(q)$$

Now,

$$\sum_{q=0}^{n} \frac{q}{t} P(q) = \frac{1}{t} \sum_{q=0}^{n} q\,P(q) = \frac{Q}{t}, \qquad \text{where Q is the mean queue at the input of f}$$

Combining the two sums from k to n, we get:

$$m = \frac{Q}{t} + \sum_{q=k}^{n} \frac{(k-q)}{t} P(q)$$

or, $$m = \frac{Q_i}{t_i} - \sum_{q_i=k_i}^{n} \frac{(q_i - k_i)}{t_i} P(q_i) \qquad \ldots\ldots \text{(6) for a function } f_i$$

If equation (6) is to be useful, we now require the probability distribution $P(q_i)$ for the queue in a function f_i. To obtain this, we proceed as follows.

Probability that there are q_i transactions in f_i at any time = $P(q_i)$

$P(q_i)$ = P(q_i transactions in entire system and all q_i are in f_i)
+ P(q_i+1 transactions in system and q_i are in f_i)
+ ...
.
.
+ P(n transactions in system and q_i are in f_i)

Since there must be at least q_i transactions in the entire system if there are q_i in f_i, and there can never be more than n transactions in the system:

$$P(q_i) = \sum_{r=q_i}^{n} P(r \text{ transactions in system and } q_i \text{ are in } f_i) \quad \ldots \text{(7)}$$

Now,

$$P(q_i \text{ transactions are in } f_i \mid r \text{ transactions in system}) = \frac{P(q_i \text{ trans in } f_i \text{ and } r \text{ in system})}{P(r \text{ trans in system})} \quad \ldots (8)$$

If the probability of any transaction which is in the system being in f_i is p, then since each transaction in the system must either be in f_i or not in f_i, and there are r transactions in the system, we can write:

$$P(q_i \text{ trans in } f_i \mid r \text{ trans in system}) = \frac{r!}{q_i!(r-q_i)!} p^{q_i} (1-p)^{r-q_i}$$

which is the Binomial Distribution. Now the probability that a given transaction is in f_i is given by:

$$p = \frac{\text{Mean time transaction spends in } f_i}{\text{Mean time transaction spends in entire system}}$$

If the mean time a transaction spends in f_i is $\bar{T}_i$, then:

$$\bar{T}_i = \frac{Q_i}{m} \qquad \text{where } Q_i \text{ is the mean number of transactions in } f_i$$

Similarly, the mean time a transaction spends in the entire system is Q/m, where Q is the mean queue of transactions in the entire system.

Therefore,

$$p = \frac{Q_i}{Q}$$

hence,

$$P(q_i \text{ trans in } f_i \mid r \text{ trans in system}) = \frac{r!}{q_i!(r-q_i)!} \left(\frac{Q_i}{Q}\right)^{q_i} \left(1 - \frac{Q_i}{Q}\right)^{r-q_i} \quad \ldots (9)$$

We now need an expression for the probability that there are r transactions in the entire system. This is fairly easily obtained.

Let the probability that any given channel has a transaction in system = p

Since there are N channels simultaneously originating calls, or, more generally, the mean number of channels which simultaneously have a transaction in the system is N, and the total number of channels is n, then:

$$p = \frac{N}{n} = \frac{Q}{n} \qquad \text{since } N = Q$$

Since any channel either has a transaction in the system or has not, and there are n channels, each of which can have at most one transaction in the system at any time, we can write:

$$P(r \text{ transactions in system}) = \frac{n!}{r!\,(n-r)!} \left(\frac{Q}{n}\right)^{r} \left(1 - \frac{Q}{n}\right)^{n-r} \quad \ldots . (10)$$

that is, the queue in the whole system is binomially distributed.

Substituting (9) and (10) into (8) and re-arranging, we get:

$$P(q_i \text{ in } f_i \text{ and } r \text{ in system}) = \frac{n!}{q_i!\,(r-q_i)!\,(n-r)!}\left(\frac{Q_i}{Q}\right)^{q_i}\left(1-\frac{Q_i}{Q}\right)^{r-q_i}\left(\frac{Q}{n}\right)^{r}\left(1-\frac{Q}{n}\right)^{n-r}$$

Substituting this into (7), we obtain our expression for $P(q_i)$

$$P(q_i) = \sum_{r=q_i}^{n} \frac{n!}{q_i!\,(r-q_i)!\,(n-r)!}\left(\frac{Q_i}{Q}\right)^{q_i}\left(1-\frac{Q_i}{Q}\right)^{r-q_i}\left(\frac{Q}{n}\right)^{r}\left(1-\frac{Q}{n}\right)^{n-r} \quad \ldots\ldots \quad (11)$$

We note that at saturation, when Q=n and r=n, the above expression reduces to the binomial distribution. Denoting quantities at saturation by the subscript T, and observing that:

$$\lim_{\substack{Q\to n \\ r\to n}} \left(1 - \frac{Q}{n}\right)^{n-r} = 1$$

we get:

$$P(q_{iT}) = \frac{n!}{q_{iT}!\,(n-q_{iT})!}\left(\frac{Q_{iT}}{n}\right)^{q_{iT}}\left(1-\frac{Q_{iT}}{n}\right)^{n-q_{iT}}$$

Substituting (11) into (6) we obtain our expression for m:

$$m = \frac{Q_i}{t_i} - \sum_{q_i=k_i}^{n}\left\{\left(\frac{q_i-k_i}{t_i}\right)\sum_{r=q_i}^{n}\left[\frac{n!}{q_i!\,(r-q_i)!\,(n-r)!}\left(\frac{Q_i}{Q}\right)^{q_i}\left(1-\frac{Q_i}{Q}\right)^{r-q_i}\left(\frac{Q}{n}\right)^{r}\left(1-\frac{Q}{n}\right)^{n-r}\right]\right\}$$

Equation (12)

B.3.5 Solution of Equations

Of course, the above expression for m involves Q_i, which is still an unknown. However, since there are three functions in series in IAS, we know that the queue in the entire system, Q, is the sum of the queues in the separate functions:

$$Q = Q_1 + Q_2 + Q_3 \quad \ldots\ldots \quad (13)$$

Equation (12), therefore, represents three equations in m, Q_1, Q_2, and Q_3, one for each function. Using (12) and (13) we can, in principle, express each of the three Q_i *in terms of m and constants only.* We can then use (13) to express Q in terms of m and constants only, and re-arrange to give m in terms of Q and constants only. Since $Q = nR_MT_C$, we could then obtain an expression for m in terms of R_M and T_C, and could substitute this, along with the expressions for Q, x, and F_{OK} into (2) to give a relation between R_M and T_C. This could then be used with equation (1) to obtain the saturation curves.

Although this is possible in principle, it is not analytically feasible. However, we have sufficient equations to make a numerical solution possible. For this purpose it is simpler to combine (2), (4) and (5), and to replace $R_M T_c$ by Q/n in the result. When this is done we have the following set of equations:

$$R_M = \frac{4R_s}{4 + T_c^2 R_s^2}$$

$$T_c = \frac{Q}{m}\left[\left(1 - \frac{Q}{n}\right)\left(1 - \frac{1}{n}\right)^Q (x_0 - x_B)\right] + x_B + \left(1 - \frac{Q}{n}\right)\left(1 - \frac{1}{n}\right)^Q (T_R + T_T)$$

$$m = \frac{Q_i}{t_i} - \sum_{q_i=k_i}^{n}\left\{\left(\frac{q_i - k_i}{t_i}\right)\sum_{r=q_i}^{n}\left[\frac{n!}{q_i!\,(r-q_i)!\,(n-r)!}\left(\frac{Q_i}{Q}\right)^{q_i}\left(1-\frac{Q_i}{Q}\right)^{r-q_i}\left(\frac{Q}{n}\right)^{r}\left(1-\frac{Q}{n}\right)^{n-r}\right]\right\}$$

for i = 1, 2, 3

$$Q = nR_M T_c$$

$$Q = Q_1 + Q_2 + Q_3$$

From these we can obtain four simultaneous equations in R_s, R_M, Q_1, Q_2 and Q_3, which must be solved numerically to give R_M as a function of R_s, so enabling us to plot the saturation curves.

B.4 Saturation

We can show that the mean achieved call rate cannot exceed a particular value irrespective of how much we increase the set mean call attempt rate.

From (6) we know that $Q = nR_M T_c$

Putting this into (1), we obtain:

$$R_M = \frac{R_s}{2} \pm \frac{R_s}{2n}\sqrt{n^2 - Q^2} \quad \ldots\ldots \quad T_c <= \frac{2}{R_s} \quad \ldots\ldots \quad (14)$$

There is evidently no real R_M for $Q > n$, so that Q must always lie in the range $0 <= Q <= n$.

Since Q is the <u>mean</u> number of transactions in the IAS queue, and the actual queue length, q, cannot be greater than n, (because of the compelled signalling), then when $Q = n$, q is <u>always</u> equal to n. In this case the proportion of time which each function spends idling is as small as it can be, that is, IAS is working as fast as it can and is said to be <u>saturated</u>.

$Q = n$ implies that $nR_M T_c = n$, or $R_M T_c = 1$ at saturation.

Putting Q=n in (14) we obtain $R_M = \frac{R_s}{2}$ at saturation

Since $R_M T_c = 1$, $T_c = \frac{1}{R_M}$

Therefore, from the above, $T_c = \frac{2}{R_s}$ at saturation.

Thus, for values of R_s greater than that required to cause saturation, $T_c > 2/R_s$, and equation (1), and hence equation (14) is no longer valid.

From all this we can infer that when R_s is such that IAS is just saturated,

$$R_M = \frac{R_s}{2}$$

The experimental data shown in figs 6.3 (a) to (e) tend to agree with this as far as can be observed, except that in a few cases there is an anomalous "bump" near the point at which the saturation curve becomes flat, which makes it difficult to tell exactly where saturation first occurs.

Putting $R_M T_c = 1$ into (5), we obtain $\alpha \; x_B$.

We might expect this result, since at saturation all channels are originating calls as fast as the rate at which IAS can respond will permit. The off-hook factor is therefore very high indeed, and virtually all the calls will end BUSY.

Putting Q=n and $F_{OK} = 0$ into equation (2), we get:

$$T_{c\ sat} = \frac{nx_B}{m_t}$$

Since $R_M T_c = 1$ at saturation, $R_{M\ sat} = 1/T_{c\ sat}$

Therefore $R_{M\ sat} = \frac{m_t}{nx_B}$

or $m_t = nx_B R_{M\ sat}$

This result is also expected, since R_M is the mean achieved call rate in calls per channel per minute, and therefore nR_M is the corresponding mean rate in calls per minute. If the mean number of transactions per call is x, then the corresponding transaction processing rate is nxR_M transactions per minute. Since at saturation all calls end busy, the transaction processing rate at saturation is $nx_B R_{M\ sat}$.

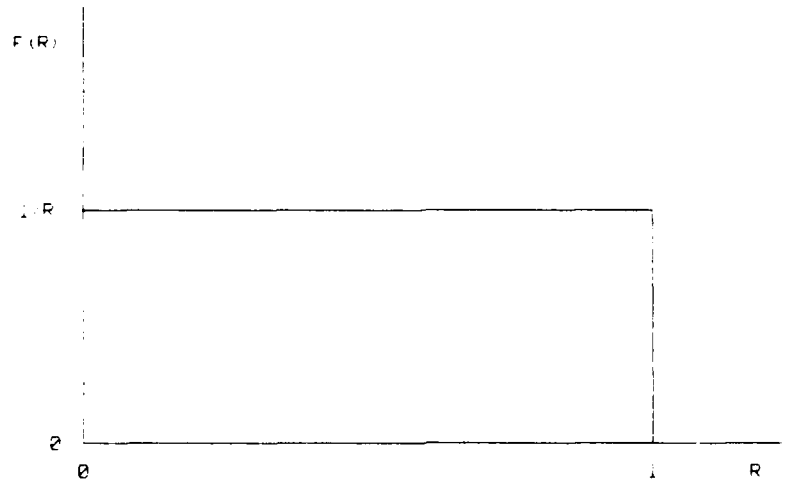


FIG B 1 RECTANGULAR DISTRIBUTION OF A RANDOM NUMBER

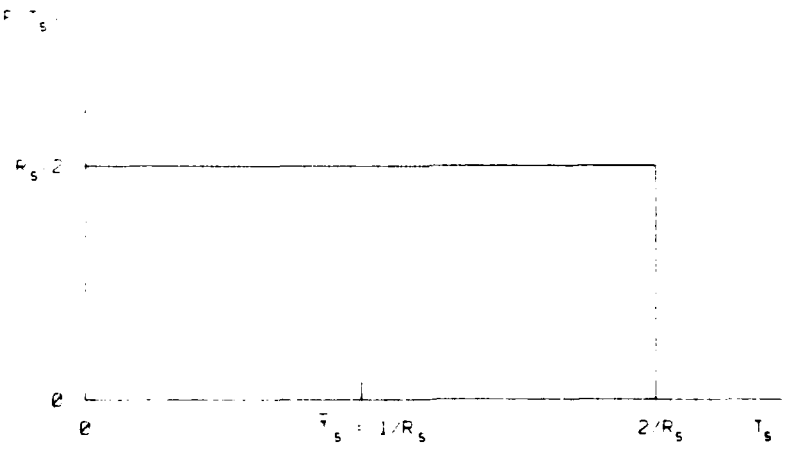


FIG B 2 PROBABILITY DENSITY FUNCTION FOR T_s ($T_c << 2/R_s$)

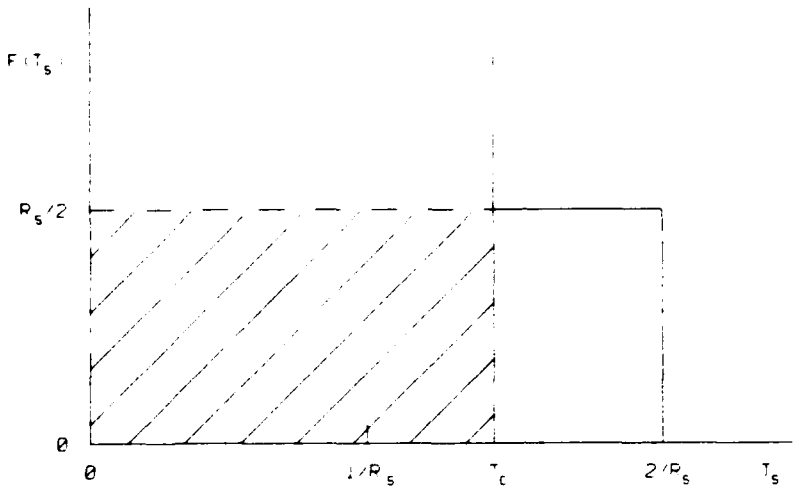


FIG B 3 PROBABILITY DENSITY FUNCTION FOR T_s WHEN T_c IS NOT NEGLIGIBLE

APPENDIX C

Statistical Variation in Achieved Mean Call Rate

In order to plot the saturation curves for IAS from the experimental results, we require the statistical variation of the measured mean call rate, R_M, to be sufficiently small for the plotted (R_M, R_S) points to lie on a smooth curve. We calculate R_M as:

$$R_M = \frac{1}{\bar{T}_s}$$

where $\bar{T}_s$ is the mean time between successive call starts on any given channel. Evidently the more calls we make, the more values of T_s we will have, and the more accurately we can compute its mean value. The purpose of this appendix is to calculate how many calls we need to make in order to be able to compute R_M to some desired accuracy.

In plotting the saturation curves, we are dealing with values of R_M in the range 0 to 30 calls per channel per minute. We choose an accuracy of ± 0.05 calls per channel per minute for R_M. R_S is, of course, a parameter which we set, and for practical purposes is therefore known exactly. We first need to know the accuracy to which we must compute the mean value of T_s in order to obtain R_M to within ± 0.05 calls per channel per minute.

Differentiating the above expression for $\bar{T}_s$ we obtain:

$$\frac{d\bar{T}_s}{dR_M} = -\frac{1}{R_M^2}$$

Using deltas to indicate small finite changes in R_M and $\bar{T}_s$, we can write:

$$\Delta\bar{T}_s = -\frac{1}{R_M^2}\ \Delta R_M$$

Thus if $\Delta R_M = 0.05$, $\Delta\bar{T}_s = -\frac{1}{R_M^2} \times 0.05$

$\Delta\bar{T}_s$ will therefore be greatest for small R_M. The smallest value of R_M we encounter experimentally is 1.

Therefore the largest uncertainty we can allow in $\bar{T}_s$ is 0.05

We now calculate the number of values of T_s we must measure. As stated in Appendix B, the distribution of T_s is rectangular, but modified by the finite minimum response time of the IAS, as shown in fig. B.3. However, the Central Limit Theorem[8] tells us that if we take a sufficiently large number of samples, we can compute a confidence interval for the mean of this distribution as if it were a Normal distribution. This we now do.

If we compute, from the measured values of T_s, a mean time between calls, and call this computed value t_s, then the actual mean time will, for some suitably chosen k, lie in the interval

$$t_s - k < \bar{T}_s < t_s + k$$

with a probability of 99%. The dependence of the length of the confidence interval L (=2k) on the number of calls, M, is given in fig. C.1 (from ref. 11). L is in units of σ, the standard deviation of the distribution of T_s, so that we need to compute a numerical value for σ. This can be done as follows.

By definition we have:

$$\sigma^2 = \int_{-\infty}^{\infty} (T_s - \bar{T}_s)^2 \, f(T_s) \, dT_s$$

For the probability density function shown in fig. B.3, this becomes:

$$\sigma^2 = \int_{T_c}^{2/R_s} (T_s - \bar{T}_s)^2 \left[\frac{R_s}{2} + \frac{R_s T_c}{2} \, \delta(T_s - T_c) \right] dT_s$$

and hence:

$$\sigma^2 = \int_{T_c}^{2/R_s} (T_s - \bar{T}_s)^2 \, \frac{R_s}{2} \, dT_s + \frac{R_s T_c}{2} \, (\bar{T}_s - T_c)^2$$

Using elementary integration, and writing $\frac{1}{R_M}$ for $\bar{T}_s$, we obtain:

$$\sigma^2 = \frac{1}{3}\left(R_s T_c^3 + \frac{4}{R_s^2} \right) - \frac{1}{R_M}\left(R_s T_c^2 - \frac{2}{R_s} \right) + \frac{1}{R_M^2}\left(1 - R_s T_c \right) \quad . \quad . \quad (C.1)$$

We can use equation (C.1) to compute σ for various values of R_s, provided that we have corresponding sets of values for R_M, R_s, and T_c. We already have the (R_M, R_s) pairs used to plot the theoretical saturation curves, and we can compute T_c using a re-arranged equation (B.1):

$$T_c = \frac{2}{R_s}\left(\frac{R_s}{R_M} - 1\right)$$

The results of these computations, for one processor supporting the CLP function, are shown in table C1 below.

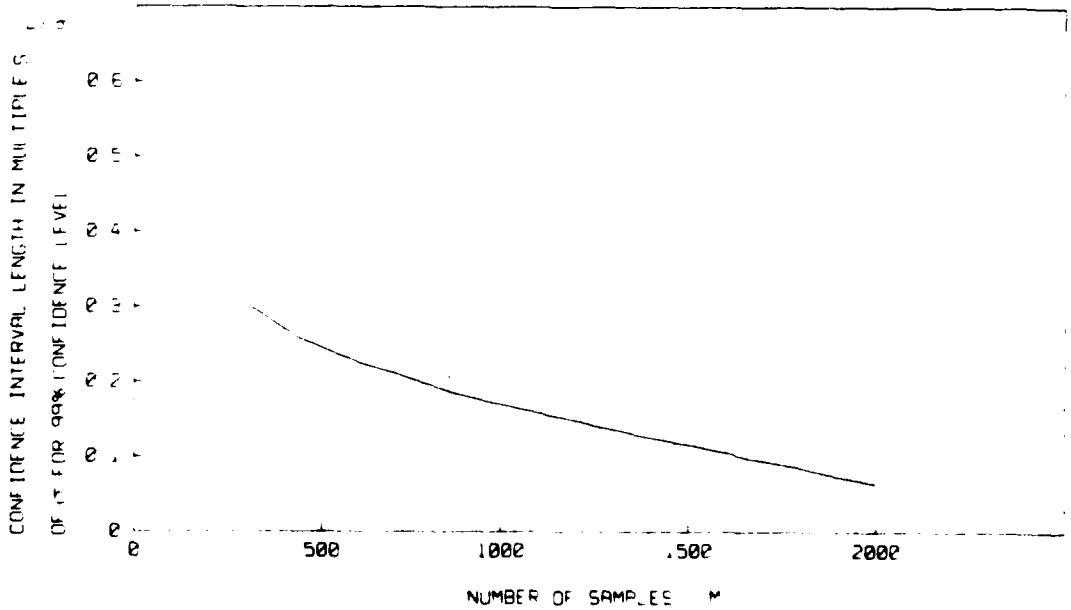
R_s	R_M	T_c	σ
1.0013	1.0	0.0724	2.0615
2.0134	2.0	0.0813	1.0141
3.0648	3.0	0.0959	0.6553
4.0162	3.8	0.1189	0.4870
5.1333	4.4	0.1591	0.3549
6.0058	4.6	0.1841	0.2695

Table C1

As can be seen, the largest values of σ occur at low values of R_s, (and hence of R_M). Taking R_M=1 as a worst case, we have σ = 2.0615. Since the required confidence interval length is L = 2k = 2 x 0.05 = 0.1, we have L/σ = 0.0485. From figure C.1 we can see that the corresponding value of M is about 2000, i.e. we need to make about 2000 measurements of T_s in order to compute the mean time between calls to within ±0.05 of its theoretical value, with a confidence level of 99%. For higher values of R_M we need fewer measurements, but in fact runs of about 2000 calls were used for all values of R_M.

If more than one processor is used to support the CLP function, the value of R_s corresponding to R_M=1 will be lower, so that σ will be lower, and L/σ will therefore be higher. Less than 2000 calls will therefore be necessary to obtain the required confidence level. The maximum variance of the T_s distribution will therefore occur when R_M=1 and only one CLP processor is used.

FIG C.1 DEPENDENCE OF CONFIDENCE INTERVAL LENGTH ON NUMBER OF SAMPLES FOR 99% CONFIDENCE LEVEL



DOCUMENT CONTROL SHEET

Overall security classification of sheet .....Unclassified.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Report 83004		u/c

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	Royal Signals and Radar Establishment
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title
Intermediate Access Switch - A Multiprocessor Application

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Connelly, T G	Griffiths, M P			

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement
Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract Design options for the implementation of the control software for a computer controlled, circuit switching exchange (Intermediate Access Switch or IAS) on a multi-processor system (DISCUS) are outlined. The impact of DISCUS and its operating system on software structure and software development, integration, and testing methods is discussed, and a set of guidelines for the division of a complete software package among several co-operating processors is drawn up. The advantages and penalties of the multi-processor approach, as compared with the single processor approach, are pointed out, and a quantitative assessment of system performance based on the use of a traffic generator is presented.

END

FILMED

DTIC